INTERNATIONAL STANDARD

ISO/IEC 16262

Third edition
2011-06.15

Information technology — Programming languages, their environments and system software interfaces — ECMAScript language specification

Technologies de l'information — Langages de programmation, leurs environnements et interfaces de logiciel système — Spécification du langage ECMAscript

Reference number
ISO/IEC 16262:2011(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 16262 was prepared by Ecma International (as ECMA-262) and was adopted, under a special “fast-track procedure”, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

This third edition cancels and replaces the second edition (ISO/IEC 16262:2002), which has been technically revised.

Introduction

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

That International Standard was submitted to ISO/IEC JTC 1 for adoption under the fast-track procedure, and approved as ISO/IEC 16262, first edition, in April 1998.

The second edition of this International 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 internationalization facilities and future language growth. The second edition of the ECMAScript standard was published as ISO/IEC 16262 in June 2002.

Since publication of the second edition of ISO/IEC 16262:2002, 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 third edition of ECMAScript. Although that work was not completed and not published as a new edition of ECMAScript, it informs continuing evolution of the language. The present third edition of ISO/IEC 16262 (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.

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

Information technology — Programming languages, their environments and system software interfaces — ECMAScript language specification

1 Scope

This International 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 International Standard.

A conforming implementation of this International Standard shall interpret characters in conformance with the Unicode Standard, Version 3.0 or later, and ISO/IEC 10646 with either UCS-2 or UTF-16 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300. If the adopted encoding form is not otherwise specified, it presumed to be the UTF-16 encoding form.

A conforming implementation of ECMAScript is permitted to provide additional types, values, objects, properties, and functions beyond those described in this International Standard. In particular, a conforming implementation of ECMAScript is permitted to provide properties not described in this International Standard, and values for those properties, for objects that are described in this International Standard.

A conforming implementation of ECMAScript is permitted to support program and regular expression syntax not described in this International Standard. 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 International Standard.

3 Normative references

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

ISO/IEC 10646, *Information technology ð Universal Coded Character Set (UCS)*

4 Overview

This clause 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 *host* 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.

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 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 can provide core scripting capabilities for a variety of host environments, and therefore the core scripting language is specified in this document apart from any particular host environment.

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.

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.

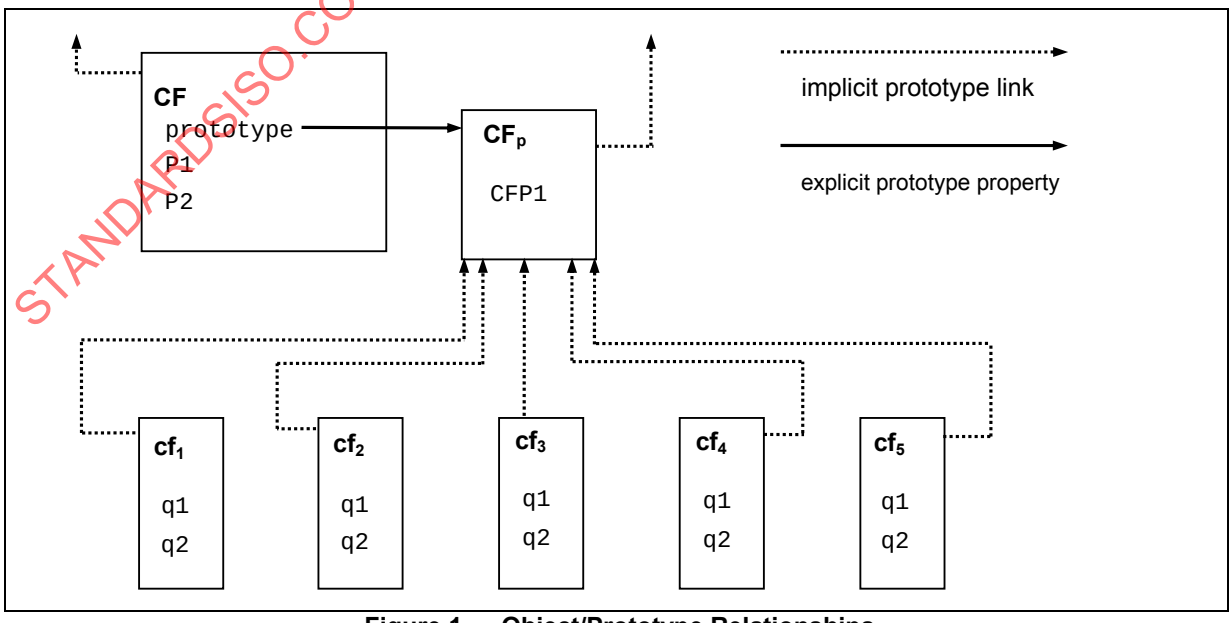

Figure 1 — Object/Prototype Relationships

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, and structure, behaviour and state are all inherited.

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

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

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 $\mathbf{cf_1}$, $\mathbf{cf_2}$, $\mathbf{cf_3}$, $\mathbf{cf_4}$, and $\mathbf{cf_5}$ by assigning a new value to the property in $\mathbf{CF_p}$.

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

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

4.3.7
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. Every built-in object is a native object. A *built-in constructor* is a built-in object that is also a constructor.

4.3.8
host object
object supplied by the host environment to complete the execution environment of ECMAScript

NOTE Any object that is not native is a host object.

4.3.9
undefined value
primitive value used when a variable has not been assigned a value

4.3.10
Undefined type
type whose sole value is the undefined value

4.3.11
null value
primitive value that represents the intentional absence of any object value

4.3.12
Null type
type whose sole value is the null value

4.3.13
Boolean value
member of the Boolean type

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

4.3.14
Boolean type
type consisting of the primitive values **true** and **false**

4.3.15
Boolean object
member of the Object type that is an instance of the standard built-in **`Boolean`** constructor

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

4.3.16
String value
primitive value that is a finite ordered sequence of zero or more 16-bit unsigned integers

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.

4.3.17
String type
set of all possible String values

4.3.18
String object
member of the Object type that is an instance of the standard built-in **`String`** constructor

NOTE A String object is created by using the **`String`** constructor in a **`new`** expression, supplying a String value as an argument. The resulting object has an internal 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).

4.3.19
Number value
primitive value corresponding to a double-precision 64-bit binary format IEEE 754 value

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

4.3.20
Number type
set of all possible Number values including the special “Not-a-Number” (NaN) values, positive infinity, and negative infinity

4.3.21
Number object
member of the Object type that is an instance of the standard built-in **`Number`** constructor

NOTE A Number object is created by using the **`Number`** constructor in a **`new`** expression, supplying a Number value as an argument. The resulting object has an internal 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).

4.3.22
Infinity
number value that is the positive infinite Number value

4.3.23
NaN
number value that is an IEEE 754 “Not-a-Number” value

4.3.24
function
member of the Object type that is an instance of the standard built-in **`Function`** constructor and that may be invoked as a subroutine

NOTE In addition to its named 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.

4.3.25
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.

4.3.26
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.

4.3.27
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.

4.3.28
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.

4.3.29
attribute
internal value that defines some characteristic of a property

4.3.30
own property
property that is directly contained by its object

4.3.31
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.

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 *Program*, that describe how sequences of tokens can form syntactically correct ECMAScript programs.

When a stream of characters is to be parsed as an ECMAScript program, 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 program is syntactically in error if the tokens in the stream

of input elements cannot be parsed as a single instance of the goal nonterminal *Program*, with no tokens left over.

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

The syntactic grammar as presented in clauses 11, 12, 13 and 14 is actually not a complete account of which token sequences are accepted as correct ECMAScript programs. 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.

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 program exactly as written. 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.

Nonterminal symbols are shown in *italic* type. The definition of a nonterminal 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 (; *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 program between the **throw** token and the *Expression*.

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

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 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.

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.

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*".

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

1. Top-level step
    a. Substep.
    b. Substep
        i. Subsubstep.
        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*) yields the absolute value of *x*, which is –*x* if *x* is negative (less than zero) and otherwise is *x* itself.

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

The 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) yields the largest integer (closest to positive infinity) that is not larger than x.

NOTE floor(x) = x−(x modulo 1).

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

6 Source Text

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 C (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 of 16-bit code units for the purposes of this specification. Such a source text may include sequences of 16-bit code units that are not valid UTF-16 character encodings. If an actual source text is encoded in a form other than 16-bit code units it must be processed as if it was first converted to UTF-16.

Syntax

SourceCharacter **::**
any Unicode code unit

Throughout the rest of this document, the phrase "code unit" and the word "character" will be used to refer to a 16-bit unsigned value used to represent a single 16-bit unit of text. The phrase "Unicode character" will be used to refer to the abstract linguistic or typographical unit represented by a single Unicode scalar value (which may be longer than 16 bits and thus may be represented by more than one code unit). 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, and identifiers, any character (code unit) may also be expressed as a Unicode escape sequence consisting of six characters, namely **\u** plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within a string literal or regular expression literal, the Unicode escape sequence contributes one character to the value of the literal. Within an identifier, the escape sequence contributes one character to the identifier.

NOTE Although this document sometimes refers to a "transformation" between a "character" within a "string" and the 16-bit unsigned integer that is the code unit of that character, there is actually no transformation because a "character" within a "string" is actually represented using that 16-bit unsigned value.

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 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 character to the String value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

7 Lexical Conventions

The source text of an ECMAScript program 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 two goal symbols for the lexical grammar. The *InputElementDiv* symbol is used in those syntactic grammar contexts where a leading division (`/`) or division-assignment (`/=`) operator is permitted. The *InputElementRegExp* symbol is used in other syntactic grammar contexts.

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

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

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 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 Unit Value	*Name*	*Formal Name*	*Usage*
\u200C	Zero width non-joiner	<ZWNJ>	*IdentifierPart*
\u200D	Zero width joiner	<ZWJ>	*IdentifierPart*
\uFEFF	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 also occur within a *StringLiteral* or a *RegularExpressionLiteral* (where they are considered significant characters forming part of the literal value) or 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 Unit Value	*Name*	*Formal Name*
\u0009	Tab	<TAB>
\u000B	Vertical Tab	<VT>
\u000C	Form Feed	<FF>
\u0020	Space	<SP>
\u00A0	No-break space	<NBSP>
\uFEFF	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 3.0. 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.

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*. 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 Unit Value	*Name*	*Formal Name*
\u000A	Line Feed	<LF>
\u000D	Carriage Return	<CR>
\u2028	Line separator	<LS>
\u2029	Paragraph separator	<PS>

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

Because a single-line comment can contain any 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

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

7.6 Identifier Names and Identifiers

Identifier Names are tokens that are interpreted according to the grammar given in the "Identifiers" section of chapter 5 of the Unicode standard, with some small modifications. An *Identifier* is an *IdentifierName* that is not a *ReservedWord* (see 7.6.1). The Unicode identifier grammar is based on both no rmative and informative character categories specified by the Unicode Standard. The characters in the specified categories in version 3.0 of the Unic ode standard must be treated as in those categories by all conforming ECMAScript implementations.

This standard specifies specific character additions: The dollar sign (**$**) and the underscore (_) are permitted anywhere in an *IdentifierName*.

Unicode escape sequences are also permitted in an *IdentifierName*, where they contribute a single character to the *IdentifierName*, as computed by the CV of the *UnicodeEscapeSequence* (see 7.8.4). The \ preceding the *UnicodeEscapeSequence* does not contribute a character 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 its *UnicodeEscapeSequence*'s CV, the result must still be a valid *IdentifierName* that has the exact same sequence of characters as the original *IdentifierName*. All interpretations of identifiers 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). The intent is that the incoming source text has been converted to normalised form C before it reaches the compiler.

ECMAScript implementations may recognise identifier characters defined in later editions of the Unicode Standard. If portability is a concern, programmers should only employ identifier characters defined in Unicode 3.0.

Syntax

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

IdentifierName **::**
IdentifierStart
IdentifierName IdentifierPart

IdentifierStart **::**
UnicodeLetter
$
_
\ *UnicodeEscapeSequence*

IdentifierPart **::**
IdentifierStart
UnicodeCombiningMark
UnicodeDigit
UnicodeConnectorPunctuation
<ZWNJ>
<ZWJ>

UnicodeLetter **::**
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 **::**
any character in the Unicode categories “Non-spacing mark (Mn)” or “Combining spacing mark (Mc)”

UnicodeDigit **::**
any character in the Unicode category “Decimal number (Nd)”

UnicodeConnectorPunctuation **::**
any character in the Unicode category “Connector punctuation (Pc)”

The definitions of the nonterminal *UnicodeEscapeSequence* is given in 7.8.4

7.6.1 Reserved Words

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

Syntax

ReservedWord **::**
Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

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`	`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`	

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.

Syntax

FutureReservedWord **:: one of**

`class`	`enum`	`extends`	`super`
`const`	`export`	`import`	

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

7.7 Punctuators

Syntax

Punctuator **:: one of**

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

DivPunctuator **:: one of**

`/`	`/=`

7.8 Literals

Syntax

Literal **::**
NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral
RegularExpressionLiteral

7.8.1 Null Literals

Syntax

NullLiteral **::**
`null`

Semantics

The value of the null literal **`null`** is the sole value of the Null type, namely **null**.

7.8.2 Boolean Literals

Syntax

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

Semantics

The value of the Boolean literal **`true`** is a value of the Boolean type, namely **true**.

The value of the Boolean literal **`false`** is a value of the Boolean type, namely **false**.

7.8.3 Numeric Literals

Syntax

NumericLiteral **::**
DecimalLiteral
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*

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

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

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

Semantics

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* :: *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** is 0.
- The MV of *DecimalDigit* **::** **1** or of *NonZeroDigit* **::** **1** or of *HexDigit* **::** **1** is 1.
- The MV of *DecimalDigit* **::** **2** or of *NonZeroDigit* **::** **2** or of *HexDigit* **::** **2** is 2.
- The MV of *DecimalDigit* **::** **3** or of *NonZeroDigit* **::** **3** or of *HexDigit* **::** **3** is 3.
- The MV of *DecimalDigit* **::** **4** or of *NonZeroDigit* **::** **4** or of *HexDigit* **::** **4** is 4.
- The MV of *DecimalDigit* **::** **5** or of *NonZeroDigit* **::** **5** or of *HexDigit* **::** **5** is 5.
- The MV of *DecimalDigit* **::** **6** or of *NonZeroDigit* **::** **6** or of *HexDigit* **::** **6** is 6.
- The MV of *DecimalDigit* **::** **7** or of *NonZeroDigit* **::** **7** or of *HexDigit* **::** **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 *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 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.

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.

7.8.4 String Literals

A string literal is zero or more characters enclosed in single or double quotes. Each character may 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.

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

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*

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

Semantics

A string literal stands for a value of the String type. The String value (SV) of the literal is described in terms of character values (CV) contributed by the various parts of the string literal. As part of this process, some 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 character sequence.
- The SV of *StringLiteral* **:: ''** is the empty character 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 character, the CV of *DoubleStringCharacter*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter DoubleStringCharacters* is a sequence of the CV of *DoubleStringCharacter* followed by all the characters in the SV of *DoubleStringCharacters* in order.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter* is a sequence of one character, the CV of *SingleStringCharacter*.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter* SingleStringCharacters is a sequence of the CV of *SingleStringCharacter* followed by all the characters in the SV of *SingleStringCharacters* in order.
- The SV of *LineContinuation* :: **** *LineTerminatorSequence* is the empty character sequence.
- The CV of *DoubleStringCharacter* :: *SourceCharacter* **but not one of " or \ or** *LineTerminator* is the *SourceCharacter* character itself.
- 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 *SourceCharacter* character itself.
- 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 a <NUL> character (Unicode value 0000).
- 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	**\u0008**	backspace	<BS>
\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>
\"	**\u0022**	double quote	"
\'	**\u0027**	single quote	'
****	**\u005C**	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 *SourceCharacter* character itself.
- The CV of *HexEscapeSequence* :: **x** *HexDigit HexDigit* is the character whose code unit value 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 character whose code unit value 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*.

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.

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**.

7.8.5 Regular Expression Literals

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 Strings of characters comprising the *RegularExpressionBody* and the *RegularExpressionFlags* are passed uninterpreted to the regular expression constructor, which interprets them according to its own, more stringent grammar. An implementation may extend the regular expression constructor's grammar, but it must not extend the *RegularExpressionBody* and *RegularExpressionFlags* productions or the productions used by these productions.

Syntax

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

RegularExpressionBody **::**
    RegularExpressionFirstChar RegularExpressionChars

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

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

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

RegularExpressionBackslashSequence **::**
    **** *RegularExpressionNonTerminator*

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

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

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

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

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

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

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 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*. 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 (Clause 16).

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 program 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 program 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 *Program*, then a semicolon is automatically inserted at the end of the input stream.

3. When, as the program is parsed from left to right, a token is encountered that is allowed by some production of the grammar, but the production is a *restricted production* and the token would be the first token for a terminal or nonterminal immediately following the annotation "[no *LineTerminator* here]" within the restricted production (and therefore such a token is called a restricted token), and the restricted token is separated from the previous token by at least one *LineTerminator*, then a semicolon is automatically inserted before the restricted token.

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

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.

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, Property Identifier, Lexical Environment, and Environment Record. 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.

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.

8.1 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**.

8.2 The Null Type

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

8.3 The Boolean Type

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

8.4 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 code unit value (see Clause 6). 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 position 0, the next element (if any) at position 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.

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

NOTE The rationale behind this design was to keep the implementation of Strings as simple and high-performing as possible. The intent is that textual data coming into the execution environment from outside (e.g., user input, text read from a file or received over the network, etc.) be converted to Unicode Normalised Form C before the running program sees it. Usually this would occur at the same time incoming text is converted from its original character encoding to Unicode (and would impose no additional overhead). Since it is recommended that ECMAScript source code be in Normalised Form C, string literals are guaranteed to be normalised (if source text is guaranteed to be normalised), as long as they do not contain any Unicode escape sequences.

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

The other 18437736874454810624 (that is, $2^{64}-2^{53}$) 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 +∞; if -2^{1024} was chosen, replace it with –∞; 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.

8.6 The Object Type

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

- A *named data property* associates a name with an ECMAScript language value and a set of Boolean attributes.
- A *named accessor property* associates a name 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.
- An *internal property* has no name and is not directly accessible via ECMAScript language operators. Internal properties exist purely for specification purposes.

There are two kinds of access for named (non-internal) properties: *get* and *put*, corresponding to retrieval and assignment, respectively.

8.6.1 Property Attributes

Attributes are used in this specification to define and explain the state of named properties. A named data property associates a name with the attributes listed in Table 5

Table 5 — Attributes of a Named Data Property

Attribute Name	*Value Domain*	*Description*
[[Value]]	Any ECMAScript language type	The value retrieved by reading the property.
[[Writable]]	Boolean	If **false**, attempts by ECMAScript code to change the property's [[Value]] attribute using [[Put]] 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]]) will fail.

A named accessor property associates a name with the attributes listed in Table 6.

Table 6 — Attributes of a Named 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 return 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 value of an attribute is not explicitly specified by this specification for a named property, 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**

8.6.2 Object Internal Properties and Methods

This specification uses various internal properties to define the semantics of object values. These internal properties are not part of the ECMAScript language. They are defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon internal properties in the manner described here. The names of internal properties are enclosed in double

square brackets [[ ]]. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a **TypeError** exception is thrown.

The Table 8 summarises the internal properties used by this specification that are applicable to all ECMAScript objects. The Table 9 summarises the internal properties used by this specification that are only applicable to some ECMAScript objects. The descriptions in these tables indicate their behaviour for native ECMAScript objects, unless stated otherwise in this document for particular kinds of native ECMAScript objects. Host objects may support these internal properties with any implementation-dependent behaviour as long as it is consistent with the specific host object restrictions stated in this document.

The "Value Type Domain" columns of 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 ECMAScript language type. "*primitive*" means Undefined, Null, Boolean, String, or Number. "*SpecOp*" means the internal property is an internal method, an implementation provided procedure 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 describes the type of the parameter. If a "SpecOp" returns a value, its parameter list is followed by the symbol "→" and the type of the returned value.

Table 8 — Internal Properties Common to All Objects

Internal Property	*Value Type Domain*	*Description*
[[Prototype]]	Object *or* Null	The prototype of this object.
[[Class]]	String	A String value indicating a specification defined classification of objects.
[[Extensible]]	Boolean	If **true**, own properties may be added to the object.
[[Get]]	SpecOp(*propertyName*) → any	Returns the value of the named property.
[[GetOwnProperty]]	SpecOp (*propertyName*) → Undefined *or* Property Descriptor	Returns the Property Descriptor of the named own property of this object, or **undefined** if absent.
[[GetProperty]]	SpecOp (*propertyName*) → Undefined *or* Property Descriptor	Returns the fully populated Property Descriptor of the named property of this object, or **undefined** if absent.
[[Put]]	SpecOp (*propertyName*, *any, Boolean*)	Sets the specified named property to the value of the second parameter. The flag controls failure handling.
[[CanPut]]	SpecOp (*propertyName*) → Boolean	Returns a Boolean value indicating whether a [[Put]] operation with *PropertyName* can be performed.
[[HasProperty]]	SpecOp (*propertyName*) → *Boolean*	Returns a Boolean value indicating whether the object already has a property with the given name.
[[Delete]]	SpecOp (*propertyName, Boolean*) → Boolean	Removes the specified named own property from the object. The flag controls failure handling.
[[DefaultValue]]	SpecOp (*Hint*) → *primitive*	Hint is a String. Returns a default value for the object.
[[DefineOwnProperty]]	SpecOp (*propertyName, PropertyDescriptor, Boolean*) → Boolean	Creates or alters the named own property to have the state described by a Property Descriptor. The flag controls failure handling.

Every object (including host objects) must implement all of the internal properties listed in Table 8. However, the [[DefaultValue]] internal method may, for some objects, simply throw a **TypeError** exception.

All objects have an internal property called [[Prototype]]. The value of this property is either **null** or an object and is used for implementing inheritance. Whether or not a native object can have a host object as its

[[Prototype]] depends on the implementation. Every [[Prototype]] chain must have finite length (that is, starting from any object, recursively accessing the [[Prototype]] internal property must eventually lead to a **null** value). Named 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 put access. Named accessor properties are inherited for both get access and put access.

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

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

The value of the [[Class]] internal property is defined by this specification for every kind of built-in object. The value of the [[Class]] internal property of a host object may be any String value except one of **`"Arguments"`**, **`"Array"`**, **`"Boolean"`**, **`"Date"`**, **`"Error"`**, **`"Function"`**, **`"JSON"`**, **`"Math"`**, **`"Number"`**, **`"Object"`**, **`"RegExp"`**, and **`"String"`**. The value of a [[Class]] internal property is used internally to distinguish different kinds of objects. Note that this specification does not provide any means for a program to access that value except through **`Object.prototype.toString`** (see 15.2.4.2).

Unless otherwise specified, the common internal methods of native ECMAScript objects behave as described in 8.12. Array objects have a slightly different implementation of the [[DefineOwnProperty]] internal method (see 15.4.5.1) and String objects have a slightly different implementation of the [[GetOwnProperty]] internal method (see 15.5.5.2). Arguments objects (10.6) have different implementations of [[Get]], [[GetOwnProperty]], [[DefineOwnProperty]], and [[Delete]]. Function objects (15.3) have a different implementation of [[Get]].

Host objects may implement these internal methods in any manner unless specified otherwise; for example, one possibility is that [[Get]] and [[Put]] for a particular host object indeed fetch and store property values but [[HasProperty]] always generates **false**. However, if any specified manipulation of a host object's internal properties is not supported by an implementation, that manipulation must throw a **TypeError** exception when attempted.

The [[GetOwnProperty]] internal method of a host object must conform to the following invariants for each property of the host 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 value of the host object's [[Extensible]] internal property 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 a host object must not permit the addition of a new property to a host object if the [[Extensible]] internal property of that host object has been observed by ECMAScript code to be **false**.

If the [[Extensible]] internal property of that host object has been observed by ECMAScript code to be **false** then it must not subsequently become **true**.

Table 9 — Internal Properties Only Defined for Some Objects

Internal Property	*Value Type Domain*	*Description*
[[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]].
[[Construct]]	SpecOp(a List of *any*) → Object	Creates an object. Invoked via the **new** operator. The arguments to the SpecOp are the arguments passed to the **new** operator. Objects that implement this internal method are called *constructors*.
[[Call]]	SpecOp(*any*, a List of *any*) → *any* or Reference	Executes code associated with the object. Invoked via a function call expression. The arguments to the SpecOp are this object and a list containing the arguments passed to the function call expression. Objects that implement this internal method are *callable*. Only callable objects that are host objects may return Reference values.
[[HasInstance]]	SpecOp(*any*) → Boolean	Returns a Boolean value indicating whether the argument is likely an Object that was constructed by this object. Of the standard built-in ECMAScript objects, only Function objects implement [[HasInstance]].
[[Scope]]	Lexical Environment	A lexical environment that defines the environment in which a Function object is executed. Of the standard built-in ECMAScript objects, only Function objects implement [[Scope]].
[[FormalParameters]]	List of Strings	A possibly empty List containing the identifier Strings of a Function's *FormalParameterList*. Of the standard built-in ECMAScript objects, only Function objects implement [[FormalParameterList]].
[[Code]]	ECMAScript code	The ECMAScript code of a function. Of the standard built-in ECMAScript objects, only Function objects implement [[Code]].
[[TargetFunction]]	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 [[TargetFunction]] 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 using the standard built-in Function.prototype.bind method. Only ECMAScript objects created using Function.prototype.bind have a [[BoundArguments]] internal property.
[[Match]]	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 [[Match]].
[[ParameterMap]]	Object	Provides a mapping between the properties of an arguments object (see 10.6) and the formal parameters of the associated function. Only ECMAScript objects that are arguments objects have a [[ParameterMap]] internal property.

8.7 The Reference Specification Type

The Reference type is used to explain the behaviour of such operators as **`delete`**, **`typeof`**, and the assignment operators. For example, the left-hand operand of an assignment is expected to produce a reference. The behaviour of assignment could, instead, be explained entirely in terms of a case analysis on the syntactic form of the left-hand operand of an assignment operator, but for one difficulty: function calls are permitted to return references. This possibility is admitted purely for the sake of host objects. No built-in ECMAScript function defined by this specification returns a reference and there is no provision for a user-defined function to return a reference. (Another reason not to use a syntactic case analysis is that it would be lengthy and awkward, affecting many parts of the specification.)

A **Reference** is a resolved name 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.

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 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.

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

8.7.1 GetValue (V)

1. If Type(*V*) is not Reference, return *V*.
2. Let *base* be the result of calling GetBase(*V*).
3. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
4. If IsPropertyReference(*V*), then
    a. If HasPrimitiveBase(*V*) is **false**, then let *get* be the [[Get]] internal method of *base*, otherwise let *get* be the special [[Get]] internal method defined below.
    b. Return the result of calling the *get* internal method using *base* as its **this** value, and passing GetReferencedName(*V*) for the argument.
5. 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.

The following [[Get]] internal method is used by GetValue when *V* is a property reference with a primitive base value. It is called using *base* as its **this** value and with property *P* as its argument. The following steps are taken:

1. Let *O* be ToObject(*base*).
2. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
3. If *desc* is **undefined**, return **undefined**.
4. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
5. Otherwise, IsAccessorDescriptor(*desc*) must be **true** so, let *getter* be *desc*.[[Get]] (see 8.10).
6. If *getter* is **undefined**, return **undefined**.
7. Return the result calling the [[Call]] internal method of *getter* providing *base* as the **this** value and providing no arguments.

NOTE The object that may be created in step 1 is not accessible outside of the above method. An implementation might choose to avoid the actual creation of the object. The only situation where such an actual property access that uses this internal method can have visible effect is when it invokes an accessor function.

8.7.2 PutValue (V, W)

1. If Type(*V*) is not Reference, throw a **ReferenceError** exception.
2. Let *base* be the result of calling GetBase(*V*).
3. If IsUnresolvableReference(*V*), then
    a. If IsStrictReference(*V*) is **true**, then
        i. Throw **ReferenceError** exception.
    b. Call the [[Put]] internal method of the global object, passing GetReferencedName(*V*) for the property name, *W* for the value, and **false** for the *Throw* flag.
4. Else if IsPropertyReference(*V*), then
    a. If HasPrimitiveBase(*V*) is **false**, then let *put* be the [[Put]] internal method of *base*, otherwise let *put* be the special [[Put]] internal method defined below.
    b. Call the *put* internal method using *base* as its **this** value, and passing GetReferencedName(*V*) for the property name, *W* for the value, and IsStrictReference(*V*) for the *Throw* flag.
5. Else *base* must be a reference whose base is an environment record. So,
    a. Call the SetMutableBinding (10.2.1) concrete method of *base*, passing GetReferencedName(*V*), *W*, and IsStrictReference(*V*) as arguments.
6. Return.

The following [[Put]] internal method is used by PutValue when *V* is a property reference with a primitive base value. It is called using *base* as its **this** value and with property *P*, value *W*, and Boolean flag *Throw* as arguments. The following steps are taken:

1. Let *O* be ToObject(*base*).
2. If the result of calling the [[CanPut]] internal method of *O* with argument *P* is **false**, then
    a. If *Throw* is **true**, then throw a **TypeError** exception.
    b. Else return.
3. Let *ownDesc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
4. If IsDataDescriptor(*ownDesc*) is **true**, then
    a. If *Throw* is **true**, then throw a **TypeError** exception.
    b. Else return.
5. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with argument *P*. This may be either an own or inherited accessor property descriptor or an inherited data property descriptor.
6. If IsAccessorDescriptor(*desc*) is **true**, then
    a. Let *setter* be *desc*.[[Set]] (see 8.10) which cannot be **undefined**.
    b. Call the [[Call]] internal method of *setter* providing *base* as the **this** value and an argument list containing only *W*.
7. Else, this is a request to create an own property on the transient object *O*
    a. If *Throw* is **true**, then throw a **TypeError** exception.
8. Return.

NOTE The object that may be created in step 1 is not accessible outside of the above method. An implementation might choose to avoid the actual creation of that transient object. The only situations where such an actual property assignment that uses this internal method can have visible effect are when it either invokes an accessor function or is in violation of a *Throw* predicated error check. When *Throw* is **true** any property assignment that would create a new property on the transient object throws an error.

8.8 The List 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.

8.9 The Completion Specification Type

The Completion type is used to explain the behaviour of statements (**`break`**, **`continue`**, **`return`** and **`throw`**) that perform nonlocal transfers of control. Values of the Completion type are triples of the form (*type*, *value*, *target*), where *type* is one of **normal**, **break**, **continue**, **return**, or **throw**, *value* is any ECMAScript language value or **empty**, and *target* is any ECMAScript identifier or **empty**. If *cv* is a completion value then *cv*.type, *cv*.value, and *cv*.target may be used to directly refer to its constituent values.

The term “abrupt completion” refers to any completion with a *type* other than **normal**.

8.10 The Property Descriptor and Property Identifier Specification Types

The Property Descriptor type is used to explain the manipulation and reification of named property attributes. Values of the Property Descriptor type are records composed of named fields where each field’s name is an attribute name and its value is a corresponding attribute value as specified in 8.6.1. 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 8.6.1 Table 5 or Table 6.

For notational convenience within this specification, an object literal-like syntax can be used to define a property descriptor value. For example, Property Descriptor {[[Value]]: 42, [[Writable]]: **false**, [[Configurable]]: **true**} defines a data property descriptor. 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 Property Descriptor. For example, if D is a property descriptor then D.[[Value]] is shorthand for “the field of D named [[Value]]”.

The Property Identifier type is used to associate a property name with a Property Descriptor. Values of the Property Identifier type are pairs of the form (name, descriptor), where name is a String and descriptor is a Property Descriptor value.

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

8.10.1 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**.

8.10.2 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**.

8.10.3 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**.

8.10.4 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. Let *obj* be the result of creating a new object as if by the expression **new Object()** where **Object** is the standard built-in constructor with that name.
3. If IsDataDescriptor(*Desc*) is **true**, then
    a. Call the [[DefineOwnProperty]] internal method of *obj* with arguments "**`value`**", Property Descriptor {[[Value]]: *Desc*.[[Value]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    b. Call the [[DefineOwnProperty]] internal method of *obj* with arguments "**`writable`**", Property Descriptor {[[Value]]: *Desc*.[[Writable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
4. Else, IsAccessorDescriptor(*Desc*) must be **true**, so
    a. Call the [[DefineOwnProperty]] internal method of *obj* with arguments "**`get`**", Property Descriptor {[[Value]]: *Desc*.[[Get]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    b. Call the [[DefineOwnProperty]] internal method of *obj* with arguments "**`set`**", Property Descriptor {[[Value]]: *Desc*.[[Set]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
5. Call the [[DefineOwnProperty]] internal method of *obj* with arguments "**`enumerable`**", Property Descriptor {[[Value]]: *Desc*.[[Enumerable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
6. Call the [[DefineOwnProperty]] internal method of *obj* with arguments "**`configurable`**", Property Descriptor {[[Value]]: *Desc*.[[Configurable]], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
7. Return *obj*.

8.10.5 ToPropertyDescriptor (Obj)

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

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

c. Set the [[Set]] field of *desc* to *setter*.
9. 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.
10. Return *desc*.

8.11 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.

8.12 Algorithms for Object Internal Methods

In the following algorithm descriptions, assume *O* is a native ECMAScript object, *P* is a String, *Desc* is a Property Description record, and *Throw* is a Boolean flag.

8.12.1 [[GetOwnProperty]] (P)

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

1. If *O* doesn't have an own property with name *P*, return **undefined**.
2. Let *D* be a newly created Property Descriptor with no fields.
3. Let *X* be *O*'s own property named *P*.
4. 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
5. 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.
6. Set *D*.[[Enumerable]] to the value of *X*'s [[Enumerable]] attribute.
7. Set *D*.[[Configurable]] to the value of *X*'s [[Configurable]] attribute.
8. Return *D*.

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

8.12.2 [[GetProperty]] (P)

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

1. Let *prop* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
2. If *prop* is not **undefined**, return *prop*.
3. Let *proto* be the value of the [[Prototype]] internal property of *O*.
4. If *proto* is **null**, return **undefined**.
5. Return the result of calling the [[GetProperty]] internal method of *proto* with argument *P*.

8.12.3 [[Get]] (P)

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

1. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
2. If *desc* is **undefined**, return **undefined**.
3. If IsDataDescriptor(*desc*) is **true**, return *desc*.[[Value]].
4. Otherwise, IsAccessorDescriptor(*desc*) must be true so, let *getter* be *desc*.[[Get]].
5. If *getter* is **undefined**, return **undefined**.
6. Return the result calling the [[Call]] internal method of *getter* providing *O* as the **this** value and providing no arguments.

8.12.4 [[CanPut]] (P)

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

1. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
2. If *desc* is not **undefined**, then
    a. If IsAccessorDescriptor(*desc*) is **true**, then
        i. If *desc*.[[Set]] is **undefined**, then return **false**.
        ii. Else return **true**.
    b. Else, *desc* must be a DataDescriptor so return the value of *desc*.[[Writable]].
3. Let *proto* be the [[Prototype]] internal property of *O*.
4. If *proto* is **null**, then return the value of the [[Extensible]] internal property of *O*.
5. Let *inherited* be the result of calling the [[GetProperty]] internal method of *proto* with property name *P*.
6. If *inherited* is **undefined**, return the value of the [[Extensible]] internal property of *O*.
7. If IsAccessorDescriptor(*inherited*) is **true**, then
    a. If *inherited*.[[Set]] is **undefined**, then return **false**.
    b. Else return **true**.
8. Else, *inherited* must be a DataDescriptor
    a. If the [[Extensible]] internal property of *O* is **false**, return **false**.
    b. Else return the value of *inherited*.[[Writable]].

Host objects may define additional constraints upon [[Put]] operations. If possible, host objects should not allow [[Put]] operations in situations where this definition of [[CanPut]] returns false.

8.12.5 [[Put]] (P, V, Throw)

When the [[Put]] internal method of *O* is called with property *P*, value *V*, and Boolean flag *Throw*, the following steps are taken:

1. If the result of calling the [[CanPut]] internal method of *O* with argument *P* is **false**, then
    a. If *Throw* is **true**, then throw a **TypeError** exception.
    b. Else return.
2. Let *ownDesc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *P*.
3. If IsDataDescriptor(*ownDesc*) is **true**, then
    a. Let *valueDesc* be the Property Descriptor {[[Value]]: *V*}.
    b. Call the [[DefineOwnProperty]] internal method of *O* passing *P*, *valueDesc*, and *Throw* as arguments.
    c. Return.
4. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with argument *P*. This may be either an own or inherited accessor property descriptor or an inherited data property descriptor.
5. If IsAccessorDescriptor(*desc*) is **true**, then
    a. Let *setter* be *desc*.[[Set]] which cannot be **undefined**.
    b. Call the [[Call]] internal method of *setter* providing *O* as the **this** value and providing *V* as the sole argument.
6. Else, create a named data property named *P* on object *O* as follows
    a. Let *newDesc* be the Property Descriptor
    {[[Value]]: *V*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.
    b. Call the [[DefineOwnProperty]] internal method of *O* passing *P*, *newDesc*, and *Throw* as arguments.
7. Return.

8.12.6 [[HasProperty]] (P)

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

1. Let *desc* be the result of calling the [[GetProperty]] internal method of *O* with property name *P*.
2. If *desc* is **undefined**, then return **false**.
3. Else return **true**.

8.12.7 [[Delete]] (P, Throw)

When the [[Delete]] internal method of *O* is called with property name *P* and the Boolean flag *Throw*, the following steps are taken:

1. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
2. If *desc* is **undefined**, then return **true**.
3. If *desc*.[[Configurable]] is **true**, then
    a. Remove the own property with name *P* from *O*.
    b. Return **true**.
4. Else if *Throw*, then throw a **TypeError** exception.
5. Return **false**.

8.12.8 [[DefaultValue]] (hint)

When the [[DefaultValue]] internal method of *O* is called with hint String, the following steps are taken:

1. Let *toString* be the result of calling the [[Get]] internal method of object *O* with argument "**`toString`**".
2. If IsCallable(*toString)* is **true** then,
    a. Let *str* be the result of calling the [[Call]] internal method of *toString*, with *O* as the **this** value and an empty argument list.
    b. If *str* is a primitive value, return *str*.
3. Let *valueOf* be the result of calling the [[Get]] internal method of object *O* with argument "**`valueOf`**".
4. If IsCallable(*valueOf)* is **true** then,
    a. Let *val* be the result of calling the [[Call]] internal method of *valueOf*, with *O* as the this value and an empty argument list.
    b. If *val* is a primitive value, return *val*.
5. Throw a **TypeError** exception.

When the [[DefaultValue]] internal method of *O* is called with hint Number, the following steps are taken:

1. Let *valueOf* be the result of calling the [[Get]] internal method of object *O* with argument "**`valueOf`**".
2. If IsCallable(*valueOf)* is **true** then,
    a. Let *val* be the result of calling the [[Call]] internal method of *valueOf*, with *O* as the **this** value and an empty argument list.
    b. If *val* is a primitive value, return *val*.
3. Let *toString* be the result of calling the [[Get]] internal method of object *O* with argument "**`toString`**".
4. If IsCallable(*toString)* is **true** then,
    a. Let *str* be the result of calling the [[Call]] internal method of *toString*, with *O* as the this value and an empty argument list.
    b. If *str* is a primitive value, return *str*.
5. Throw a **TypeError** exception.

When the [[DefaultValue]] internal method of *O* is called with no hint, then it behaves as if the hint were Number, unless *O* is a Date object (see 15.9.6), in which case it behaves as if the hint were String.

The above specification of [[DefaultValue]] for native objects can return only primitive values. If a host object implements its own [[DefaultValue]] internal method, it must ensure that its [[DefaultValue]] internal method can return only primitive values.

8.12.9 [[DefineOwnProperty]] (P, Desc, Throw)

In the following algorithm, the term "Reject" means "If *Throw* is **true**, then throw a **TypeError** exception, otherwise return **false**". The 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**.

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

1. Let *current* be the result of calling the [[GetOwnProperty]] internal method of *O* with property name *P*.
2. Let *extensible* be the value of the [[Extensible]] internal property of *O*.
3. If *current* is **undefined** and *extensible* is **false**, then Reject.
4. If *current* is **undefined** and *extensible* is **true**, then
    a. If IsGenericDescriptor(*Desc*) or IsDataDescriptor(*Desc*) is **true**, then
        i. 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.
    b. Else, *Desc* must be an accessor Property Descriptor so,
        i. 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.
    c. Return **true**.
5. Return **true**, if every field in *Desc* is absent.
6. 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).
7. If the [[Configurable]] field of *current* is **false** then
    a. Reject, if the [[Configurable]] field of *Desc* is **true**.
    b. Reject, if the [[Enumerable]] field of *Desc* is present and the [[Enumerable]] fields of *current* and *Desc* are the Boolean negation of each other.
8. If IsGenericDescriptor(*Desc*) is **true**, then no further validation is required.
9. Else, if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) have different results, then
    a. Reject, if the [[Configurable]] field of *current* is **false**.
    b. If IsDataDescriptor(*current*) is **true**, then
        i. 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. 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.
10. Else, if IsDataDescriptor(*current*) and IsDataDescriptor(*Desc*) are both **true**, then
    a. If the [[Configurable]] field of *current* is **false**, then
        i. Reject, 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. Reject, 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.
11. Else, IsAccessorDescriptor(*current*) and IsAccessorDescriptor(*Desc*) are both **true** so,
    a. If the [[Configurable]] field of *current* is **false**, then
        i. Reject, if the [[Set]] field of *Desc* is present and SameValue(*Desc*.[[Set]], *current*.[[Set]]) is **false**.
        ii. Reject, if the [[Get]] field of *Desc* is present and SameValue(*Desc*.[[Get]], *current*.[[Get]]) is **false**.
12. 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.
13. Return **true**.

However, if *O* is an Array object, it has a more elaborate [[DefineOwnProperty]] internal method defined in 15.4.5.1.

NOTE Step 10.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**.

9 Type Conversion and Testing

The ECMAScript runtime system performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion abstract operations. These abstract operations are not a part of the language; they are defined here to aid the specification of the semantics of the language. The conversion abstract operations are polymorphic; that is, they can accept a value of any ECMAScript language type, but not of specification types.

9.1 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 10:

Table 10 — ToPrimitive Conversions

Input Type	*Result*
Undefined	The result equals the *input* argument (no conversion).
Null	The result equals the *input* argument (no conversion).
Boolean	The result equals the *input* argument (no conversion).
Number	The result equals the *input* argument (no conversion).
String	The result equals the *input* argument (no conversion).
Object	Return a default value for the Object. The default value of an object is retrieved by calling the [[DefaultValue]] internal method of the object, passing the optional hint *PreferredType*. The behaviour of the [[DefaultValue]] internal method is defined by this specification for all native ECMAScript objects in 8.12.8.

9.2 ToBoolean

The abstract operation ToBoolean converts its argument to a value of type Boolean according to Table 11:

Table 11 — ToBoolean Conversions

Argument Type	*Result*
Undefined	**false**
Null	**false**
Boolean	The result equals the input argument (no conversion).
Number	The result is **false** if the argument is **+0**, **–0**, or **NaN**; otherwise the result is **true**.
String	The result is **false** if the argument is the empty String (its length is zero); otherwise the result is **true**.
Object	**true**

9.3 ToNumber

The abstract operation ToNumber converts its argument to a value of type Number according to Table 12:

Table 12 — To Number Conversions

Argument Type	*Result*
Undefined	**NaN**
Null	**+0**
Boolean	The result is **1** if the argument is **true**. The result is **+0** if the argument is **false**.
Number	The result equals the input argument (no conversion).
String	See grammar and note below.
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(*input argument*, hint Number). 2. Return ToNumber(*primValue*).

9.3.1 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

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**.

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.

9.4 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. If *number* is **NaN**, return **+0**.
3. If *number* is **+0**, **–0**, **+∞**, or **–∞**, return *number*.
4. Return the result of computing sign(*number*) × floor(abs(*number*)).

9.5 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. If *number* is **NaN**, **+0**, **–0**, +∞, or –∞, return **+0**.
3. Let *posInt* be sign(*number*) * floor(abs(*number*)).
4. Let *int32bit* be *posInt* modulo 2^{32}; 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* and k is mathematically an integer multiple of 2^{32}.
5. If *int32bit* is greater than or equal to 2^{31}, return *int32bit* – 2^{32}, otherwise return *int32bit*.

NOTE Given the above definition of ToInt32:

- The ToInt32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
- ToInt32(ToUint32(x)) is equal to ToInt32(x) for all values of x. (It is to preserve this latter property that +∞ and –∞ are mapped to **+0**.)
- ToInt32 maps **–0** to **+0**.

9.6 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 abstraction operation functions as follows:

1. Let *number* be the result of calling ToNumber on the input argument.
2. If *number* is **NaN**, +0, –0, +∞, or –∞, return **+0**.
3. Let *posInt* be sign(*number*) × floor(abs(*number*)).
4. Let *int32bit* be *posInt* modulo 2^{32}; 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* and k is mathematically an integer multiple of 2^{32}.
5. Return *int32bit*.

NOTE Given the above definition of ToUint32:

- Step 5 is the only difference between ToUint32 and ToInt32.
- The ToUint32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves that value unchanged.
- ToUint32(ToInt32(x)) is 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**.

9.7 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. If *number* is **NaN**, +0, –0, +∞, or –∞, return **+0**.
3. Let *posInt* be sign(*number*) × floor(abs(*number*)).
4. Let *int16bit* be *posInt* modulo 2^{16}; 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* and *k* is mathematically an integer multiple of 2^{16}.
5. Return *int16bit*.

NOTE Given the above definition of ToUint16:

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

9.8 ToString

The abstract operation ToString converts its argument to a value of type String according to Table 13:

Table 13 — ToString Conversions

Argument Type	*Result*
Undefined	**"undefined"**
Null	**"null"**
Boolean	If the argument is **true**, then the result is **"true"**. If the argument is **false**, then the result is **"false".**
Number	See 9.8.1.
String	Return the input argument (no conversion)
Object	Apply the following steps: 1. Let *primValue* be ToPrimitive(input argument, hint String). 2. Return ToString(*primValue*).

9.8.1 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 infinity, 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**'.
7. If $0 < n \leq 21$, 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 $-6 < n \leq 0$, 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 $k \geq 1$, $10^{k-1} \leq s < 10^k$, the Number value for $s \times 10^{n-k}$ is *m*, and *k* is as small as possible. If there are multiple possibilities for *s*, choose the value of *s* for which $s \times 10^{n-k}$ is closest in value to *m*. If there are two such possible values of *s*, choose the one that is even. Note that *k* is the number of digits in the decimal representation of *s* and that *s* is not divisible by 10.

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

> 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://cm.bell-labs.com/netlib/fp/dtoa.c.gz and as http://cm.bell-labs.com/netlib/fp/g_fmt.c.gz and may also be found at the various **`netlib`** mirror sites.

9.9 ToObject

The abstract operation ToObject converts its argument to a value of type Object according to Table 14:

Table 14 — ToObject

Argument Type	*Result*
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Create a new Boolean object whose [[PrimitiveValue]] internal property is set to the value of the argument. See 15.6 for a description of Boolean objects.
Number	Create a new Number object whose [[PrimitiveValue]] internal property is set to the value of the argument. See 15.7 for a description of Number objects.
String	Create a new String object whose [[PrimitiveValue]] internal property is set to the value of the argument. See 15.5 for a description of String objects.
Object	The result is the input argument (no conversion).

9.10 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 15:

Table 15 — CheckObjectCoercible Results

Argument Type	*Result*
Undefined	Throw a **TypeError** exception.
Null	Throw a **TypeError** exception.
Boolean	Return
Number	Return
String	Return
Object	Return

9.11 IsCallable

The abstract operation IsCallable determines if its argument, which must be an ECMAScript language value, is a callable function Object according to Table 16:

Table 16 — IsCallable Results

Argument Type	*Result*
Undefined	Return **false**.
Null	Return **false**.
Boolean	Return **false**.
Number	Return **false**.
String	Return **false**.
Object	If the argument object has a [[Call]] internal method, then return **true**, otherwise return **false**.

9.12 The SameValue Algorithm

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. 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 and *y* is NaN, return **true**.
    b. If *x* is +0 and *y* is -0, return **false**.
    c. If *x* is -0 and *y* is +0, return **false**.
    d. If *x* is the same Number value as *y*, return **true**.
    e. Return **false**.
5. If Type(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return **false**.
6. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**; otherwise, return **false**.
7. Return true if *x* and *y* refer to the same object. Otherwise, return **false**.

10 Executable Code and Execution Contexts

10.1 Types of Executable Code

There are three types of ECMAScript executable code:

- *Global code* is source text that is treated as an ECMAScript *Program*. The global code of a particular *Program* does not include any source text that is parsed as part of a *FunctionBody*.

- *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 *Program*. The eval code for a particular invocation of **`eval`** is the global code portion of that *Program*.

- *Function code* is source text that is parsed as part of a *FunctionBody*. The *function code* of a particular *FunctionBody* does not include any source text that is parsed as part of a nested *FunctionBody*. *Function code* also denotes the source text supplied when using the built-in **`Function`** object as a constructor. More precisely, the last parameter provided 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, all parameters except the last one are converted to Strings and concatenated together, separated by commas. The resulting String is interpreted as the *FormalParameterList* for the *FunctionBody* defined by the last parameter. The function code for a particular instantiation of a **`Function`** does not include any source text that is parsed as part of a nested *FunctionBody*.

10.1.1 Strict Mode Code

An ECMAScript *Program* syntactic unit may be processed using either unrestricted or strict mode syntax and semantics. When processed using strict mode the three types of ECMAScript code are referred to as 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).
- 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 *PropertyAssignment* is strict function code if its *FunctionDeclaration*, *FunctionExpression*, or *PropertyAssignment* 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.

10.2 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 *WithStatement*, 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 execution of the surrounding function.

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 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 *Program* and *WithStatement* that associate identifier bindings with the properties of some object.

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 two concrete subclasses, declarative environment record and object environment record. The abstract class includes the abstract specification

methods defined in Table 17. These abstract methods have distinct concrete algorithms for each of the concrete subclasses.

Table 17 — Abstract Methods of Environment Records

Method	*Purpose*
HasBinding(N)	Determine if an environment record has a binding for 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 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.
SetMutableBinding(N,V, S)	Set the value of an already existing mutable binding in a n 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 a n already existing binding fr om 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**.
ImplicitThisValue()	Returns the value to use as the **this** 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 and/or function declarations. A declarative environment record binds the set of identifiers defined by the declarations contained within its scope.

In addition to the mutable bindings supported by all Environment Records, declarative environment records also provide for immutable bindings. An immutable binding is one where the association between an identifier and a value may not be modified once it has been established. Creation and initialisation of immutable binding are distinct steps so it is possible for such bindings to exist in either an initialised or uninitialised state. Declarative environment records support the methods listed in Table 18 in addition to the Environment Record abstract specification methods:

Table 18 — Additional Methods of Declarative Environment Records

Method	*Purpose*
CreateImmutableBinding(N)	Create a new but uninitialised immutable binding in an environment record. The String value *N* is the text of the bound name.
InitializeImmutableBinding(N,V)	Set the value of an already existing but uninitialised immutable 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.

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. If it does not have such a binding, 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 initialised to the value **undefined**. 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 set its bound value to **undefined**. If *D* is true record that the newly created binding may be deleted by a subsequent DeleteBinding call.

10.2.1.1.3 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* is a mutable binding, change its bound value to *V*.
4. Else this must be an attempt to change the value of an immutable binding so if *S* if **true** throw a **TypeError** exception.

10.2.1.1.4 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 immutable binding, then
    a. If *S* is **false**, return the value **undefined**, otherwise throw a **ReferenceError** exception.
4. Else, return the value currently bound to *N* in *envRec*.

10.2.1.1.5 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* is cannot be deleted, return **false**.
4. Remove the binding for *N* from *envRec*.
5. Return **true**.

10.2.1.1.6 ImplicitThisValue()

Declarative Environment Records always return **undefined** as their ImplicitThisValue.

1. Return **undefined**.

10.2.1.1.7 CreateImmutableBinding (N)

The concrete Environment Record method CreateImmutableBinding for declarative environment records creates a new immutable binding for the name *N* that is initialised to the value **undefined**. 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.

10.2.1.1.8 InitializeImmutableBinding (N,V)

The concrete Environment Record method InitializeImmutableBinding 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 immutable 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 immutable binding for N.
3. Set the bound value for *N* in *envRec* to *V*.
4. Record that the immutable binding for *N* in *envRec* has been initialised.

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 can be configured to provide their binding object as an implicit this value for use in function calls. This capability is used to specify the behaviour of With Statement (12.10) induced bindings. The capability is controlled by a *provideThis* Boolean value that is associated with each object environment record. By default, the value of *provideThis* 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 calling the [[HasProperty]] internal method of *bindings*, passing *N* 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 calling the [[HasProperty]] internal method of *bindings*, passing *N* as the property name, is **false**.
4. If D is **true** then let *configValue* be **true** otherwise let *configValue* be **false**.
5. Call the [[DefineOwnProperty]] internal method of *bindings*, passing *N*, Property Descriptor {[[Value]]:**undefined**, [[Writable]]: **true**, [[Enumerable]]: **true** , [[Configurable]]: *configValue*}, and **true** as arguments.

10.2.1.2.3 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* should already exist 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. Call the [[Put]] internal method of *bindings* with arguments *N*, *V*, and *S*.

10.2.1.2.4 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 calling the [[HasProperty]] internal method of *bindings*, passing *N* as the property name.
4. If *value* is **false**, then
    a. If *S* is **false**, return the value **undefined**, otherwise throw a **ReferenceError** exception.
5. Return the result of calling the [[Get]] internal method of *bindings*, passing *N* for the argument.

10.2.1.2.5 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* and **false** as arguments.

10.2.1.2.6 ImplicitThisValue()

Object Environment Records return **undefined** as their ImplicitThisValue unless their *provideThis* flag is **true**.

1. Let *envRec* be the object environment record for which the method was invoked.
2. If the *provideThis* flag of *envRec* is **true**, return the binding object for *envRec*.
3. Otherwise, return **undefined**.

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*, an identifier 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 mode 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 mode 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.

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*.

10.2.3 The Global Environment

The *global environment* is a unique Lexical Environment which is created before any ECMAScript code is executed. The global env ironment's Environment Record is an object environment record whose binding object is the global object (15.1). The global environment's outer environment reference is **null**.

As ECMAScript code is executed, additional properties may be added to the global object and the initial properties may be modified.

10.3 Execution Contexts

When control is transferred to ECMAScript executable code, control is entering an *execution context*. Active execution contexts logically form a stack. The top execution context on this logical stack is the running execution context. 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 state is necessary to track the execution progress of its associated code. In addition, each execution context has the state components listed in Table 19.

Table 19 —Execution Context State Components

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* and *FunctionDeclarations* within this execution context.
ThisBinding	The value associated with the **`this`** keyword within ECMAScript code associated with 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.

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", "VariableEnvironment" and "ThisBinding" are used without qualification they are in reference to those components of the running execution context.

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

10.3.1 Identifier Resolution

Identifier resolution is the process of determining the binding of an *Identifier* using the LexicalEnvironment of the running execution context. During execution of ECMAScript code, the syntactic production *PrimaryExpression* **:** *Identifier* is evaluated using the following algorithm:

1. Let *env* be the running execution context's LexicalEnvironment.
2. If the syntactic production that is being evaluated is contained in a strict mode code, then let *strict* be **true**, else let *strict* be **false**.
3. Return the result of calling GetIdentifierReference function passing *env*, *Identifier*, and *strict* as arguments.

The result of evaluating an identifier is always a value of type Reference with its referenced name component equal to the *Identifier* String.

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 itself 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) is performed. The exact manner in which these actions occur depend on the type of code being entered.

10.4.1 Entering Global Code

The following steps are performed when control enters the execution context for global code:

1. Initialise the execution context using the global code as described in 10.4.1.1.
2. Perform Declaration Binding Instantiation as described in 10.5 using the global code.

10.4.1.1 Initial Global Execution Context

The following steps are performed to initialise a global execution context for ECMAScript code *C*:

1. Set the VariableEnvironment to the Global Environment.
2. Set the LexicalEnvironment to the Global Environment.
3. Set the ThisBinding to the global object.

10.4.2 Entering Eval Code

The following steps are performed when control enters the execution context for eval code:

1. If there is no calling context or if the eval code is not being evaluated by a direct call (15.1.2.1.1) to the eval function then,
    a. Initialise the execution context as if it was a global execution context using the eval code as *C* as described in 10.4.1.1.
2. Else,
    a. Set the ThisBinding to the same value as the ThisBinding of the calling execution context.
    b. Set the LexicalEnvironment to the same value as the LexicalEnvironment of the calling execution context.
    c. Set the VariableEnvironment to the same value as the VariableEnvironment of the calling execution context.
3. If the eval code is strict code, then
    a. Let *strictVarEnv* be the result of calling NewDeclarativeEnvironment passing the LexicalEnvironment as the argument.
    b. Set the LexicalEnvironment to *strictVarEnv*.
    c. Set the VariableEnvironment to *strictVarEnv*.
4. Perform Declaration Binding Instantiation as described in 10.5 using the eval code.

10.4.2.1 Strict Mode Restrictions

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.

10.4.3 Entering Function Code

The following steps are performed when control enters the execution context for function code contained in function object *F*, a caller provided *thisArg*, and a caller provided *argumentsList*:

1. If the function code is strict code, set the ThisBinding to *thisArg*.
2. Else if *thisArg* is **null** or **undefined**, set the ThisBinding to the global object.
3. Else if Type(*thisArg*) is not Object, set the ThisBinding to ToObject(*thisArg*).
4. Else set the ThisBinding to *thisArg*.
5. Let *localEnv* be the result of calling NewDeclarativeEnvironment passing the value of the [[Scope]] internal property of *F* as the argument.
6. Set the LexicalEnvironment to *localEnv*.

7. Set the VariableEnvironment to *localEnv*.
8. Let *code* be the value of *F*'s [[Code]] internal property.
9. Perform Declaration Binding Instantiation using the function code *code* and *argumentsList* as described in 10.5.

10.5 Declaration Binding Instantiation

Every execution context has an associated VariableEnvironment. Variables and functions declared in ECMAScript code evaluated in an execution context are added as bindings in that VariableEnvironment's Environment Record. For function code, parameters are also added as bindings to that Environment Record.

Which Environment Record is used to bind a declaration and its kind depends upon the type of ECMAScript code executed by the execution context, but the remainder of the behaviour is generic. On entering an execution context, bindings are created in the VariableEnvironment as follows using the caller provided *code* and, if it is function code, argument List *args*:

1. Let *env* be the environment record component of the running execution context's VariableEnvironment.
2. If *code* is eval code, then let *configurableBindings* be **true** else let *configurableBindings* be **false**.
3. If *code* is strict mode code, then let *strict* be **true** else let *strict* be **false**.
4. If *code* is function code, then
    a. Let *func* be the function whose [[Call]] internal method initiated execution of *code*. Let *names* be the value of *func*'s [[FormalParameters]] internal property.
    b. Let *argCount* be the number of elements in *args*.
    c. Let *n* be the number 0.
    d. For each String *argName* in *names*, in list order do
        i. Let *n* be the current value of *n* plus 1.
        ii. If *n* is greater than *argCount*, let *v* be **undefined** otherwise let *v* be the value of the *n*'th element of *args*.
        iii. Let *argAlreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *argName* as the argument.
        iv. If *argAlreadyDeclared* is **false**, call *env's* CreateMutableBinding concrete method passing *argName* as the argument.
        v. Call *env's* SetMutableBinding concrete method passing *argName*, *v*, and *strict* as the arguments.
5. For each *FunctionDeclaration f* in *code*, in source text order do
    a. Let *fn* be the *Identifier* in *FunctionDeclaration f*.
    b. Let *fo* be the result of instantiating *FunctionDeclaration f* as described in Clause 13.
    c. Let *funcAlreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *fn* as the argument.
    d. If *funcAlreadyDeclared* is **false**, call *env's* CreateMutableBinding concrete method passing *fn* and *configurableBindings* as the arguments.
    e. Else if *env* is the environment record component of the global environment then
        i. Let *go* be the global object.
        ii. Let *existingProp* be the resulting of calling the [[GetProperty]] internal method of *go* with argument *fn*.
        iii. If *existingProp* .[[Configurable]] is **true**, then
            1. Call the [[DefineOwnProperty]] internal method of *go*, passing *fn*, Property Descriptor {[[Value]]: **undefined**, [[Writable]]: **true**, [[Enumerable]]: **true** , [[Configurable]]: *configurableBindings* }, and **true** as arguments.
        iv. Else if IsAccessorDescriptor(*existingProp*) or *existingProp* does not have attribute values {[[Writable]]: **true**, [[Enumerable]]: **true**}, then
            1. Throw a TypeError exception.
    f. Call *env's* SetMutableBinding concrete method passing *fn*, *fo*, and *strict* as the arguments.
6. Let *argumentsAlreadyDeclared* be the result of calling *env's* HasBinding concrete method passing `"arguments"` as the argument.
7. If *code* is function code and *argumentsAlreadyDeclared* is **false**, then
    a. Let *argsObj* be the result of calling the abstract operation CreateArgumentsObject (10.6) passing *func*, *names*, *args*, *env* and *strict* as arguments.
    b. If *strict* is **true**, then

i. Call *env*'s CreateImmutableBinding concrete method passing the String "**`arguments`**" as the argument.
ii. Call *env*'s InitializeImmutableBinding concrete method passing "**`arguments`**" and *argsObj* as arguments.

c. Else,
i. Call *env*'s CreateMutableBinding concrete method passing the String "**`arguments`**" as the argument.
ii. Call *env*'s SetMutableBinding concrete method passing "**`arguments`**", *argsObj*, and **false** as arguments.

8. For each *VariableDeclaration* and *VariableDeclarationNoIn d* in *code*, in source text order do
a. Let *dn* be the *Identifier* in *d*.
b. Let *varAlreadyDeclared* be the result of calling *env's* HasBinding concrete method passing *dn* as the argument.
c. If *varAlreadyDeclared* is **false**, then
i. Call *env*'s CreateMutableBinding concrete method passing *dn* and *configurableBindings* as the arguments.
ii. Call *env*'s SetMutableBinding concrete method passing *dn*, **undefined**, and *strict* as the arguments.

10.6 Arguments Object

When control enters an execution context for function code, an arguments object is created unless (as specified in 10.5) the identifier **`arguments`** occurs as an *Identifier* in the function's *FormalParameterList* or occurs as the *Identifier* of a *VariableDeclaration* or *FunctionDeclaration* contained in the function code.

The arguments object is created by calling the abstract operation CreateArgumentsObject with arguments *func* the function object whose code is to be evaluated, *names* a List containing the function's formal parameter names, *args* the actual arguments passed to the [[Call]] internal method, *env* the variable environment for the function code, and *strict* a Boolean that indicates whether or not the function code is strict code. When CreateArgumentsObject is called the following steps are performed:

1. Let *len* be the number of elements in *args*.
2. Let *obj* be the result of creating a new ECMAScript object.
3. Set all the internal methods of *obj* as specified in 8.12.
4. Set the [[Class]] internal property of *obj* to "**`Arguments`**".
5. Let *Object* be the standard built-in Object constructor (15.2.2).
6. Set the [[Prototype]] internal property of *obj* to the standard built-in Object prototype object (15.2.4).
7. Call the [[DefineOwnProperty]] internal method on *obj* passing "**`length`**", the Property Descriptor {[[Value]]: *len*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}, and **false** as arguments.
8. Let *map* be the result of creating a new object as if by the expression **`new Object()`** where **`Object`** is the standard built-in constructor with that name
9. Let *mappedNames* be an empty List.
10. Let *indx* = *len* - 1.
11. Repeat while *indx* >= 0,
a. Let *val* be the element of *args* at 0-origined list position *indx*.
b. Call the [[DefineOwnProperty]] internal method on *obj* passing ToString(*indx*), the property descriptor {[[Value]]: *val*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false** as arguments.
c. If *indx* is less than the number of elements in *names*, then
i. Let *name* be the element of *names* at 0-origined list position *indx*.
ii. If *strict* is **false** and *name* is not an element of *mappedNames*, then
1. Add *name* as an element of the list *mappedNames*.
2. Let *g* be the result of calling the *MakeArgGetter* abstract operation with arguments *name* and *env*.
3. Let *p* be the result of calling the *MakeArgSetter* abstract operation with arguments *name* and *env*.
4. Call the [[DefineOwnProperty]] internal method of *map* passing ToString(*indx*), the Property Descriptor {[[Set]]: *p*, [[Get]]: *g*, [[Configurable]]: **true**}, and **false** as arguments.

d. Let *indx* = *indx* - 1

12. If *mappedNames* is not empty, then
    a. Set the [[ParameterMap]] internal property of *obj* to *map*.
    b. Set the [[Get]], [[GetOwnProperty]], [[DefineOwnProperty]], and [[Delete]] internal methods of *obj* to the definitions provided below.
13. If *strict* is **false**, then
    a. Call the [[DefineOwnProperty]] internal method on *obj* passing "`callee`", the property descriptor {[[Value]]: *func*, [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}, and **false** as arguments.
14. Else, *strict* is **true** so
    a. Let *thrower* be the [[ThrowTypeError]] function Object (13.2.3).
    b. Call the [[DefineOwnProperty]] internal method of *obj* with arguments `"caller"`, PropertyDescriptor {[[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
    c. Call the [[DefineOwnProperty]] internal method of *obj* with arguments `"callee"`, PropertyDescriptor {[[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
15. 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 *body* be the result of concatenating the Strings "`return` ", *name*, and "`;`".
2. Return the result of creating a function object as described in 13.2 using no *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 *param* be the String *name* concatenated with the String "`_arg`".
2. Let *body* be the String "**<name> = <param>;**" with **<name>** replaced by the value of *name* and **<param>** replaced by the value of *param*.
3. Return the result of creating a function object as described in 13.2 using a List containing the single String *param* 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 *map* be the value of the [[ParameterMap]] internal 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. If the value of *isMapped* is **undefined**, then
    a. Let *v* be the result of calling the default [[Get]] internal method (8.12.3) on the arguments object passing *P* as the argument.
    b. If *P* is `"caller"` and *v* is a strict mode Function object, throw a **TypeError** exception.
    c. Return *v*.
4. Else, *map* contains a formal parameter mapping for *P* so,
    a. Return the result of calling the [[Get]] internal method of *map* passing *P* as the argument.

The [[GetOwnProperty]] internal method of an arguments object for a no n-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 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 the [[Get]] internal method of *map* passing *P* as the argument.

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*, Property Descriptor *Desc*, and Boolean flag *Throw* performs the following steps:

1. Let *map* be the value of the [[ParameterMap]] internal 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 (8.12.9) on the arguments object passing *P*, *Desc*, and **false** as the arguments.
4. If *allowed* is **false**, then
    a. If *Throw* is **true** then throw a **TypeError** exception, otherwise return **false**.
5. 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*, and **false** as the arguments.
    b. Else
        i. If *Desc*.[[Value]] is present, then
            1. Call the [[Put]] internal method of *map* passing *P*, *Desc*.[[Value]], and *Throw* as the arguments.
        ii. If *Desc*.[[Writable]] is present and its value is **false**, then
            1. Call the [[Delete]] internal method of *map* passing *P* and **false** as arguments.
6. Return **true**.

The [[Delete]] internal method of an arguments object for a non-strict mode function with formal parameters when called with a property name *P* and Boolean flag *Throw* performs the following steps:

1. Let *map* be the value of the [[ParameterMap]] internal 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 *result* be the result of calling the default [[Delete]] internal method (8.12.7) on the arguments object passing *P* and *Throw* as the arguments.
4. If *result* is **true** and the value of *isMapped* is not **undefined**, then
    a. Call the [[Delete]] internal method of *map* passing *P*, and **false** as the arguments.
5. Return *result*.

NOTE 1 For non-strict mode functions the array index (defined in 15.4) named 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
ArrayLiteral
ObjectLiteral
(*Expression* **)**

11.1.1 The `this` Keyword

The **this** keyword evaluates to the value of the ThisBinding of the current execution context.

11.1.2 Identifier Reference

An *Identifier* is evaluated by performing Identifier Resolution as specified in 10.3.1. The result of evaluating an *Identifier* is always a value of type Reference.

11.1.3 Literal Reference

A *Literal* is evaluated as described in 7.8.

11.1.4 Array Initialiser

An array initialiser is an expression describing the initialisation of an Array object, written in a form of a literal. It is 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.

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*
ElementList **,** $Elision_{opt}$ *AssignmentExpression*

Elision :
,
Elision **,**

Semantics

The production *ArrayLiteral* **:** **[** $Elision_{opt}$ **]** is evaluated as follows:

1. Let *array* be the result of creating a new object as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
2. Let *pad* be the result of evaluating *Elision*; if not present, use the numeric value zero.
3. Call the [[Put]] internal method of *array* with arguments "**length**", *pad*, and **false**.
4. Return *array*.

The production *ArrayLiteral* **: [** *ElementList* **]** is evaluated as follows:

1. Return the result of evaluating *ElementList*.

The production *ArrayLiteral* **: [** *ElementList* **,** $Elision_{opt}$ **]** is evaluated as follows:

1. Let *array* be the result of evaluating *ElementList*.
2. Let *pad* be the result of evaluating *Elision*; if not present, use the numeric value zero.
3. Let *len* be the result of calling the [[Get]] internal method of *array* with argument "**length**".
4. Call the [[Put]] internal method of *array* with arguments "**length**", ToUint32(*pad+len*), and **false**.
5. Return *array*.

The production *ElementList* **:** $Elision_{opt}$ *AssignmentExpression* is evaluated as follows:

1. Let *array* be the result of creating a new object as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
2. Let *firstIndex* be the result of evaluating *Elision*; if not present, use the numeric value zero.
3. Let *initResult* be the result of evaluating *AssignmentExpression*.
4. Let *initValue* be GetValue(*initResult*).
5. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(*firstIndex)*, the Property Descriptor { [[Value]]: *initValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
6. Return *array*.

The production *ElementList* **:** *ElementList* **,** $Elision_{opt}$ *AssignmentExpression* is evaluated as follows:

1. Let *array* be the result of evaluating *ElementList*.
2. Let *pad* be the result of evaluating *Elision*; if not present, use the numeric value zero.
3. Let *initResult* be the result of evaluating *AssignmentExpression*.
4. Let *initValue* be GetValue(*initResult*).
5. Let *len* be the result of calling the [[Get]] internal method of *array* with argument "**length**".
6. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(ToUint32((*pad+len*)) and the Property Descriptor { [[Value]]: *initValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
7. Return *array*.

The production *Elision* **: ,** is evaluated as follows:

1. Return the numeric value 1.

The production *Elision* **:** *Elision* **,** is evaluated as follows:

1. Let *preceding* be the result of evaluating *Elision*.
2. Return *preceding*+1.

NOTE [[DefineOwnProperty]] is used to ensur e 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 [[Put]].

11.1.5 Object Initialiser

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 **:**
{ }
{ *PropertyNameAndValueList* **}**
{ *PropertyNameAndValueList* **, }**

PropertyNameAndValueList **:**
PropertyAssignment
PropertyNameAndValueList **,** *PropertyAssignment*

PropertyAssignment **:**
PropertyName **:** *AssignmentExpression*
get *PropertyName* **( ) {** *FunctionBody* **}**
set *PropertyName* **(** *PropertySetParameterList* **) {** *FunctionBody* **}**

PropertyName **:**
IdentifierName
StringLiteral
NumericLiteral

PropertySetParameterList **:**
Identifier

Semantics

The production *ObjectLiteral* **: { }** is evaluated as follows:

1. Return a new object created as if by the expression **new Object()** where **Object** is the standard built-in constructor with that name.

The productions *ObjectLiteral* **: {** *PropertyNameAndValueList* **}** and *ObjectLiteral* **: {** *PropertyNameAndValueList* **, }** are evaluated as follows:

1. Return the result of evaluating *PropertyNameAndValueList*.

The production *PropertyNameAndValueList* **:** *PropertyAssignment* is evaluated as follows:

1. Let *obj* be the result of creating a new object as if by the expression **new Object()** where **Object** is the standard built-in constructor with that name.
2. Let *propId* be the result of evaluating *PropertyAssignment*.
3. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId*.name, *propId*.descriptor, and **false**.
4. Return *obj*.

The production
PropertyNameAndValueList **:** *PropertyNameAndValueList* **,** *PropertyAssignment*
is evaluated as follows:

1. Let *obj* be the result of evaluating *PropertyNameAndValueList*.
2. Let *propId* be the result of evaluating *PropertyAssignment*.
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. Call the [[DefineOwnProperty]] internal method of *obj* with arguments *propId*.name, *propId*.descriptor, and **false**.
6. Return *obj*.

If the above steps would throw a **SyntaxError** then an implementation must treat the error as an early error (Clause 16).

The production *PropertyAssignment* : *PropertyName* : *AssignmentExpression* is evaluated as follows:

1. Let *propName* be the result of evaluating *PropertyName*.
2. Let *exprValue* be the result of evaluating *AssignmentExpression*.
3. Let *propValue* be GetValue(*exprValue*).
4. Let *desc* be the Property Descriptor{[[Value]]: *propValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}
5. Return Property Identifier (*propName*, *desc*).

The production *PropertyAssignment* : **get** *PropertyName* **(** **)** **{** *FunctionBody* **}** is evaluated as follows:

1. Let *propName* be the result of evaluating *PropertyName*.
2. Let *closure* be the result of creating a new Function object as specified in 13.2 with an empty parameter list and body specified by *FunctionBody*. Pass in the LexicalEnvironment of the running execution context as the *Scope*. Pass in **true** as the *Strict* flag if the *PropertyAssignment* is contained in strict code or if its *FunctionBody* is strict code.
3. Let *desc* be the Property Descriptor{[[Get]]: *closure*, [[Enumerable]]: **true**, [[Configurable]]: **true**}
4. Return Property Identifier (*propName*, *desc*).

The production *PropertyAssignment* : **set** *PropertyName* **(** *PropertySetParameterList* **)** **{** *FunctionBody* **}** is evaluated as follows:

1. Let *propName* be the result of evaluating *PropertyName*.
2. Let *closure* be the result of creating a new Function object as specified in 13.2 with parameters specified by *PropertySetParameterList* and body specified by *FunctionBody*. Pass in the LexicalEnvironment of the running execution context as the *Scope*. Pass in **true** as the *Strict* flag if the *PropertyAssignment* is contained in strict code or if its *FunctionBody* is strict code.
3. Let *desc* be the Property Descriptor{[[Set]]: *closure*, [[Enumerable]]: **true**, [[Configurable]]: **true**}
4. Return Property Identifier (*propName*, *desc*).

It is a **SyntaxError** if the *Identifier* **"eval"** or the *Identifier* **"arguments"** occurs as the *Identifier* in a *PropertySetParameterList* of a *PropertyAssignment* that is contained in strict code or if its *FunctionBody* is strict code.

The production *PropertyName* : *IdentifierName* is evaluated as follows:

1. Return the String value containing the same sequence of characters as the *IdentifierName*.

The production *PropertyName* : *StringLiteral* is evaluated as follows:

1. Return the SV of the *StringLiteral*.

The production *PropertyName* : *NumericLiteral* is evaluated as follows:

1. Let *nbr* be the result of forming the value of the *NumericLiteral*.
2. Return ToString(*nbr*).

11.1.6 The Grouping Operator

The production *PrimaryExpression* **:** **(** *Expression* **)** is evaluated as follows:

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.

11.2 Left-Hand-Side Expressions

Syntax

MemberExpression **:**
PrimaryExpression
FunctionExpression
MemberExpression **[** *Expression* **]**
MemberExpression **.** *IdentifierName*
new *MemberExpression* *Arguments*

NewExpression **:**
MemberExpression
new *NewExpression*

CallExpression **:**
MemberExpression *Arguments*
CallExpression *Arguments*
CallExpression **[** *Expression* **]**
CallExpression **.** *IdentifierName*

Arguments **:**
( )
(*ArgumentList* **)**

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

LeftHandSideExpression **:**
NewExpression
CallExpression

11.2.1 Property Accessors

Properties are accessed by name, using either the dot notation:

MemberExpression **.** *IdentifierName*
CallExpression **.** *IdentifierName*

or the bracket notation:

MemberExpression **[** *Expression* **]**
CallExpression **[** *Expression* **]**

The dot notation is explained by the following syntactic conversion:

MemberExpression **.** *IdentifierName*

is identical in its behaviour to

MemberExpression **[** <*identifier-name-string*> **]**

and similarly

CallExpression **.** *IdentifierName*

is identical in its behaviour to

CallExpression **[** <*identifier-name-string*> **]**

where <*identifier-name-string*> is a string literal containing the same sequence of characters after processing of Unicode escape sequences as the *IdentifierName*.

The production *MemberExpression* **:** *MemberExpression* **[** *Expression* **]** is evaluated as follows:

1. Let *baseReference* be the result of evaluating *MemberExpression*.
2. Let *baseValue* be GetValue(*baseReference*).
3. Let *propertyNameReference* be the result of evaluating *Expression*.
4. Let *propertyNameValue* be GetValue(*propertyNameReference*).
5. Call CheckObjectCoercible(*baseValue*).
6. Let *propertyNameString* be ToString(*propertyNameValue*).
7. If the syntactic production that is being evaluated is contained in strict mode code, let *strict* be **true**, else let *strict* be **false**.
8. Return a value of type Reference whose base value is *baseValue* and whose referenced name is *propertyNameString*, and whose strict mode flag is *strict*.

The production *CallExpression* **:** *CallExpression* **[** *Expression* **]** is evaluated in exactly the same manner, except that the contained *CallExpression* is evaluated in step 1.

11.2.2 The new Operator

The production *NewExpression* **: new** *NewExpression* is evaluated as follows:

1. Let *ref* be the result of evaluating *NewExpression*.
2. Let *constructor* be GetValue(*ref*).
3. If Type(*constructor*) is not Object, throw a **TypeError** exception.
4. If *constructor* does not implement the [[Construct]] internal method, throw a **TypeError** exception.
5. Return the result of calling the [[Construct]] internal method on *constructor*, providing no arguments (that is, an empty list of arguments).

The production *MemberExpression* **: new** *MemberExpression Arguments* is evaluated as follows:

1. Let *ref* be the result of evaluating *MemberExpression*.
2. Let *constructor* be GetValue(*ref*).
3. Let *argList* be the result of evaluating *Arguments*, producing an internal list of argument values (11.2.4).
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*, providing the list *argList* as the argument values.

11.2.3 Function Calls

The production *CallExpression* **:** *MemberExpression Arguments* is evaluated as follows:

1. Let *ref* be the result of evaluating *MemberExpression*.
2. Let *func* be GetValue(*ref*).
3. Let *argList* be the result of evaluating *Arguments*, producing an internal list of argument values (see 11.2.4).
4. If Type(*func*) is not Object, throw a **TypeError** exception.
5. If IsCallable(*func*) is **false**, throw a **TypeError** exception.
6. If Type(*ref*) is Reference, then
    a. If IsPropertyReference(*ref*) is **true**, then

i. Let *thisValue* be GetBase(*ref*).
b. Else, the base of *ref* is an Environment Record
i. Let *thisValue* be the result of calling the ImplicitThisValue concrete method of GetBase(*ref*).
7. Else, Type(*ref*) is not Reference.
a. Let *thisValue* be **undefined**.
8. Return the result of calling the [[Call]] internal method on *func*, providing *thisValue* as the **this** value and providing the list *argList* as the argument values.

The production *CallExpression* **:** *CallExpression Arguments* is evaluated in exactly the same manner, except that the contained *CallExpression* is evaluated in step 1.

NOTE The returned result will never be of type Reference if *func* is a native ECMAScript object. Whether calling a host object can return a value of type Reference is implementation-dependent. If a value of type Reference is returned, it must be a non-strict Property Reference.

11.2.4 Argument Lists

The evaluation of an argument list produces a List of values (see 8.8).

The production *Arguments* **: ( )** is evaluated as follows:

1. Return an empty List.

The production *Arguments* **: (** *ArgumentList* **)** is evaluated as follows:

1. Return the result of evaluating *ArgumentList*.

The production *ArgumentList* **:** *AssignmentExpression* is evaluated as follows:

1. Let *ref* be the result of evaluating *AssignmentExpression*.
2. Let *arg* be GetValue(*ref*).
3. Return a List whose sole item is *arg*.

The production *ArgumentList* **:** *ArgumentList* **,** *AssignmentExpression* is evaluated as follows:

1. Let *precedingArgs* be the result of evaluating *ArgumentList*.
2. Let *ref* be the result of evaluating *AssignmentExpression*.
3. Let *arg* be GetValue(*ref*).
4. 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.

11.2.5 Function Expressions

The production *MemberExpression* **:** *FunctionExpression* is evaluated as follows:

1. Return the result of evaluating *FunctionExpression*.

11.3 Postfix Expressions

Syntax

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

11.3.1 Postfix Increment Operator

The production *PostfixExpression* **:** *LeftHandSideExpression* [no *LineTerminator* here] **++** is evaluated as follows:

1. Let *lhs* be the result of evaluating *LeftHandSideExpression*.
2. 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"**
3. Let *oldValue* be ToNumber(GetValue(*lhs*)).
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. Call PutValue(*lhs*, *newValue*).
6. Return *oldValue*.

11.3.2 Postfix Decrement Operator

The production *PostfixExpression* **:** *LeftHandSideExpression* [no *LineTerminator* here] **--** is evaluated as follows:

1. Let *lhs* be the result of evaluating *LeftHandSideExpression*.
2. 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"**
3. Let *oldValue* be ToNumber(GetValue(*lhs*)).
4. Let *newValue* be the result of subtracting the value **1** from *oldValue*, using the same rules as for the **-** operator (11.6.3).
5. Call PutValue(*lhs*, *newValue*).
6. Return *oldValue*.

11.4 Unary Operators

Syntax

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

11.4.1 The `delete` Operator

The production *UnaryExpression* **:** **delete** *UnaryExpression* is evaluated as follows:

1. Let *ref* be the result of evaluating *UnaryExpression*.
2. If Type(*ref*) is not Reference, return **true**.
3. If IsUnresolvableReference(*ref*) then,
   a. If IsStrictReference(*ref*) is **true**, throw a **SyntaxError** exception.
   b. Else, return **true**.

4. If IsPropertyReference(*ref*) is **true**, then
    a. Return the result of calling the [[Delete]] internal method on ToObject(GetBase(*ref*)) providing GetReferencedName(*ref*) and IsStrictReference(*ref*) as the arguments.
5. Else, *ref* is a Reference to an Environment Record binding, so
    a. If IsStrictReference(*ref*) is **true**, throw a **SyntaxError** exception.
    b. Let *bindings* be GetBase(*ref*).
    c. 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

The production *UnaryExpression* **: `void`** *UnaryExpression* is evaluated as follows:

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Call GetValue(*expr*).
3. Return **undefined**.

NOTE GetValue must be called even though its value is not used because it may have observable side-effects.

11.4.3 The `typeof` Operator

The production *UnaryExpression* **: `typeof`** *UnaryExpression* is evaluated as follows:

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. Return a String determined by Type(*val*) according to Table 20.

Table 20 — typeof Operator Results

***Type of** val*	***Result***
Undefined	**`"undefined"`**
Null	**`"object"`**
Boolean	**`"boolean"`**
Number	**`"number"`**
String	**`"string"`**
Object (native and does not implement [[Call]])	**`"object"`**
Object (native or host and does implement [[Call]])	**`"function"`**
Object (host and does not implement [[Call]])	Implementation-defined except may not be **`"undefined"`**, **`"boolean"`**, **`"number"`**, or **`"string"`.**

11.4.4 Prefix Increment Operator

The production *UnaryExpression* **: `++`** *UnaryExpression* is evaluated as follows:

1. Let *expr* be the result of evaluating UnaryExpression.
2. 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"**

3. Let *oldValue* be ToNumber(GetValue(*expr*)).
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. Call PutValue(*expr*, *newValue*).
6. Return *newValue*.

11.4.5 Prefix Decrement Operator

The production *UnaryExpression* **: --** *UnaryExpression* is evaluated as follows:

1. Let *expr* be the result of evaluating UnaryExpression.
2. 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"**
3. Let *oldValue* be ToNumber(GetValue(*expr*)).
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. Call PutValue(*expr*, *newValue*).
6. Return *newValue*.

11.4.6 Unary + Operator

The unary + operator converts its operand to Number type.

The production *UnaryExpression* **: +** *UnaryExpression* is evaluated as follows:

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

11.4.7 Unary - Operator

The unary **-** operator converts its operand to Number type and then negates it. Note that negating **+0** produces **–0**, and negating **–0** produces **+0**.

The production *UnaryExpression* **: -** *UnaryExpression* is evaluated as follows:

1. Let *expr* be the result of evaluating UnaryExpression.
2. Let *oldValue* be ToNumber(GetValue(*expr*)).
3. If *oldValue* is **NaN**, return **NaN**.
4. Return the result of negating *oldValue*; that is, compute a Number with the same magnitude but opposite sign.

11.4.8 Bitwise NOT Operator (~)

The production *UnaryExpression* **: ~** *UnaryExpression* is evaluated as follows:

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToInt32(GetValue(*expr*)).
3. Return the result of applying bitwise complement to *oldValue*. The result is a signed 32-bit integer.

11.4.9 Logical NOT Operator (!)

The production *UnaryExpression* : **!** *UnaryExpression* is evaluated as follows:

1. Let *expr* be the result of evaluating *UnaryExpression*.
2. Let *oldValue* be ToBoolean(GetValue(*expr*)).
3. If *oldValue* is **true**, return **false**.
4. Return **true**.

11.5 Multiplicative Operators

Syntax

MultiplicativeExpression :
UnaryExpression
MultiplicativeExpression ***** *UnaryExpression*
MultiplicativeExpression **/** *UnaryExpression*
MultiplicativeExpression **%** *UnaryExpression*

Semantics

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. Let *right* be the result of evaluating UnaryExpression.
4. Let *rightValue* be GetValue(*right*).
5. Let *leftNum* be ToNumber(*leftValue*).
6. Let *rightNum* be ToNumber(*rightValue*).
7. Return the result of applying the specified operation (*, /, or %) to *leftNum* and *rightNum*. 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.
- 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 $r = n - (d \times q)$ 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*

11.6.1 The Addition operator (+)

The addition operator either performs string concatenation or numeric addition.

The production *AdditiveExpression* **:** *AdditiveExpression* **+** *MultiplicativeExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating MultiplicativeExpression.
4. Let *rval* be GetValue(*rref*).
5. Let *lprim* be ToPrimitive(*lval*).
6. Let *rprim* be ToPrimitive(*rval*).
7. If Type(*lprim*) is String or Type(*rprim*) is String, then
    a. Return the String that is the result of concatenating ToString(*lprim*) followed by ToString(*rprim*)
8. Return the result of applying the addition operation to ToNumber(*lprim*) and ToNumber(*rprim*). See the Note below 11.6.3.

NOTE 1 No hint is provided in the calls to ToPrimitive in steps 5 and 6. All native 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. Host objects may handle the absence of a hint in some other manner.

NOTE 2 Step 7 differs from step 3 of the comparison algorithm for the relational operators (11.8.5), by using the logical-or operation instead of the logical-and operation.

11.6.2 The Subtraction Operator (-)

The production *AdditiveExpression* **:** *AdditiveExpression* **-** *MultiplicativeExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating AdditiveExpression.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating MultiplicativeExpression.
4. Let *rval* be GetValue(*rref*).
5. Let *lnum* be ToNumber(*lval*).
6. Let *rnum* be ToNumber(*rval*).
7. Return the result of applying the subtraction operation to *lnum* and *rnum*. See the note below 11.6.3.

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*

11.7.1 The Left Shift Operator (<<)

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

The production *ShiftExpression* **:** *ShiftExpression* **<<** *AdditiveExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *AdditiveExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *lnum* be ToInt32(*lval*).
6. Let *rnum* be ToUint32(*rval*).
7. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
8. Return the result of left shifting *lnum* by *shiftCount* bits. The result is a signed 32-bit integer.

11.7.2 The Signed Right Shift Operator (>>)

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

The production *ShiftExpression* **:** *ShiftExpression* **>>** *AdditiveExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *AdditiveExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *lnum* be ToInt32(*lval*).
6. Let *rnum* be ToUint32(*rval*).
7. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.

8. Return the result of performing a sign-extending right shift of *lnum* by *shiftCount* bits. The most significant bit is propagated. The result is a signed 32-bit integer.

11.7.3 The Unsigned Right Shift Operator (>>>)

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

The production *ShiftExpression* **:** *ShiftExpression* **>>>** *AdditiveExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *ShiftExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *AdditiveExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *lnum* be ToUint32(*lval*).
6. Let *rnum* be ToUint32(*rval*).
7. Let *shiftCount* be the result of masking out all but the least significant 5 bits of *rnum*, that is, compute *rnum* & 0x1F.
8. Return the result of performing a zero-filling right shift of *lnum* by *shiftCount* bits. Vacated bits are filled with zero. The result is an unsigned 32-bit integer.

11.8 Relational Operators

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*

NOTE The "NoIn" variants are needed to avoid confusing the `in` operator in a relational expression with the `in` operator in a `for` statement.

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*.

11.8.1 The Less-than Operator (<)

The production *RelationalExpression* **:** *RelationalExpression* **<** *ShiftExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *r* be the result of performing abstract relational comparison *lval* < *rval*. (see 11.8.5)
6. If *r* is **undefined**, return **false**. Otherwise, return *r*.

11.8.2 The Greater-than Operator (>)

The production RelationalExpression : *RelationalExpression* **>** *ShiftExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *r* be the result of performing abstract relational comparison *rval* < *lval* with *LeftFirst* equal to **false**. (see 11.8.5).
6. If *r* is **undefined**, return **false**. Otherwise, return *r*.

11.8.3 The Less-than-or-equal Operator (<=)

The production RelationalExpression : *RelationalExpression* **<=** *ShiftExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *r* be the result of performing abstract relational comparison *rval* < *lval* with *LeftFirst* equal to **false**. (see 11.8.5).
6. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

11.8.4 The Greater-than-or-equal Operator (>=)

The production RelationalExpression : *RelationalExpression* **>=** *ShiftExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *r* be the result of performing abstract relational comparison *lval* < *rval*. (see 11.8.5)
6. If *r* is **true** or **undefined**, return **false**. Otherwise, return **true**.

11.8.5 The Abstract Relational Comparison Algorithm

The comparison *x* < *y*, where *x* and *y* are values, produces **true**, **false**, or **undefined** (which indicates that at least one operand is **NaN**). In addition to *x* and *y* the algorithm takes a Boolean flag named *LeftFirst* as a parameter. The flag is used to control the order in which operations with potentially visible side-effects are performed upon *x* and *y*. It is necessary because ECMAScript specifies left to right evaluation of expressions. The default value of *LeftFirst* is **true** and indicates that the *x* parameter corresponds to an expression that occurs to the left of the *y* parameter's corresponding expression. If *LeftFirst* is **false**, the reverse is the case and operations must be performed upon *y* before *x*. Such a comparison is performed as follows:

1. If the *LeftFirst* flag is **true**, then
    a. Let *px* be the result of calling ToPrimitive(*x*, hint Number).
    b. Let *py* be the result of calling ToPrimitive(*y*, hint Number).
2. 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. Let *px* be the result of calling ToPrimitive(*x*, hint Number).
3. If it is not the case that both Type(*px*) is String and Type(*py*) is String, then
   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**.
4. Else, both *px* and *py* are Strings
   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**.

NOTE 1 Step 3 differs from step 7 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.

11.8.6 The instanceof operator

The production *RelationalExpression*: *RelationalExpression* **`instanceof`** *ShiftExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be GetValue(*rref*).
5. If Type(*rval*) is not Object, throw a **TypeError** exception.
6. If *rval* does not have a [[HasInstance]] internal method, throw a **TypeError** exception.
7. Return the result of calling the [[HasInstance]] internal method of *rval* with argument *lval*.

11.8.7 The in operator

The production *RelationalExpression* : *RelationalExpression* **`in`** *ShiftExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *RelationalExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *ShiftExpression*.
4. Let *rval* be GetValue(*rref*).
5. If Type(*rval*) is not Object, throw a **TypeError** exception.
6. Return the result of calling the [[HasProperty]] internal method of *rval* with argument ToString(*lval*).

11.9 Equality Operators

Syntax

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

EqualityExpressionNoIn **:**
RelationalExpressionNoIn
EqualityExpressionNoIn **==** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!=** *RelationalExpressionNoIn*
EqualityExpressionNoIn **===** *RelationalExpressionNoIn*
EqualityExpressionNoIn **!==** *RelationalExpressionNoIn*

Semantics

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.

The *EqualityExpressionNoIn* productions are evaluated in the same manner as the *EqualityExpression* productions except that the contained *EqualityExpressionNoIn* and *RelationalExpressionNoIn* are evaluated instead of the contained *EqualityExpression* and *RelationalExpression*, respectively.

11.9.1 The Equals Operator (==)

The production EqualityExpression : *EqualityExpression* **==** *RelationalExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be GetValue(*rref*).
5. Return the result of performing abstract equality comparison *rval* == *lval*. (see 11.9.3).

11.9.2 The Does-not-equals Operator (!=)

The production EqualityExpression **:** *EqualityExpression* **!=** *RelationalExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *r* be the result of performing abstract equality comparison *rval* == *lval*. (see 11.9.3).
6. If *r* is **true**, return **false**. Otherwise, return **true**.

11.9.3 The Abstract Equality Comparison 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. If Type(*x*) is Undefined, return **true**.
   b. If Type(*x*) is Null, return **true**.
   c. If Type(*x*) is Number, then
      i. If *x* is **NaN**, return **false**.

ii. If *y* is **NaN**, return **false**.
iii. If *x* is the same Number value as *y*, return **true**.
iv. If *x* is **+0** and *y* is **–0**, return **true**.
v. If *x* is **–0** and *y* is **+0**, return **true**.
vi. Return **false**.

d. If Type(*x*) is String, then return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return **false**.
e. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**. Otherwise, return **false**.
f. Return **true** if *x* and *y* refer to the same object. Otherwise, return **false**.

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**.

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.

11.9.4 The Strict Equals Operator (===)

The production EqualityExpression : *EqualityExpression* **===** *RelationalExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *EqualityExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be GetValue(*rref*).
5. Return the result of performing the strict equality comparison *rval* === *lval*. (See 11.9.6)

11.9.5 The Strict Does-not-equal Operator (!==)

The production EqualityExpression : *EqualityExpression* **!==** *RelationalExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *EqualityExpression*.

2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *RelationalExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *r* be the result of performing strict equality comparison *rval* === *lval*. (See 11.9.6)
6. If *r* is **true**, return **false**. Otherwise, return **true**.

11.9.6 The Strict Equality Comparison 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 return **true** if *x* and *y* are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return **false**.
6. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**; otherwise, return **false**.
7. Return **true** if *x* and *y* refer to the same object. Otherwise, return **false**.

NOTE This algorithm differs from the SameValue Algorithm (9.12) in its treatment of signed zeroes and NaNs.

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*

Semantics

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. Let *rref* be the result of evaluating *B*.
4. Let *rval* be GetValue(*rref*).
5. Let *lnum* be ToInt32(*lval*).
6. Let *rnum* be ToInt32(*rval*).
7. 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*

LogicalANDExpressionNoIn **:**
BitwiseORExpressionNoIn
LogicalANDExpressionNoIn **&&** *BitwiseORExpressionNoIn*

LogicalORExpression **:**
LogicalANDExpression
LogicalORExpression **||** *LogicalANDExpression*

LogicalORExpressionNoIn **:**
LogicalANDExpressionNoIn
LogicalORExpressionNoIn **||** *LogicalANDExpressionNoIn*

Semantics

The production *LogicalANDExpression* **:** *LogicalANDExpression* **&&** *BitwiseORExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *LogicalANDExpression*.
2. Let *lval* be GetValue(*lref*).
3. If ToBoolean(*lval*) is **false**, return *lval*.
4. Let *rref* be the result of evaluating *BitwiseORExpression*.
5. Return GetValue(*rref*).

The production *LogicalORExpression* **:** *LogicalORExpression* **||** *LogicalANDExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. Let *lval* be GetValue(*lref*).
3. If ToBoolean(*lval*) is **true**, return *lval*.
4. Let *rref* be the result of evaluating *LogicalANDExpression*.
5. Return GetValue(*rref*).

The *LogicalANDExpressionNoIn* and *LogicalORExpressionNoIn* productions are evaluated in the same manner as the *LogicalANDExpression* and *LogicalORExpression* productions except that the 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.

11.12 Conditional Operator (? :)

Syntax

ConditionalExpression **:**
LogicalORExpression
LogicalORExpression **?** *AssignmentExpression* **:** *AssignmentExpression*

ConditionalExpressionNoIn **:**
LogicalORExpressionNoIn
LogicalORExpressionNoIn **?** *AssignmentExpression* **:** *AssignmentExpressionNoIn*

Semantics

The production *ConditionalExpression* **:** *LogicalORExpression* **?** *AssignmentExpression* **:** *AssignmentExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *LogicalORExpression*.
2. If ToBoolean(GetValue(*lref*)) is **true**, then
   a. Let *trueRef* be the result of evaluating the first *AssignmentExpression*.
   b. Return GetValue(*trueRef*).
3. Else
   a. Let *falseRef* be the result of evaluating the second *AssignmentExpression*.
   b. Return GetValue(*falseRef*).

The *ConditionalExpressionNoIn* production is evaluated in the same manner as the *ConditionalExpression* production except that the contained *LogicalORExpressionNoIn*, *AssignmentExpression* and *AssignmentExpressionNoIn* are evaluated instead 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.

11.13 Assignment Operators

Syntax

AssignmentExpression **:**
ConditionalExpression
LeftHandSideExpression **=** *AssignmentExpression*
LeftHandSideExpression AssignmentOperator AssignmentExpression

AssignmentExpressionNoIn **:**
ConditionalExpressionNoIn
LeftHandSideExpression **=** *AssignmentExpressionNoIn*
LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn

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

Semantics

The *AssignmentExpressionNoIn* productions are evaluated in the same manner as the *AssignmentExpression* productions except that the contained *ConditionalExpressionNoIn* and *AssignmentExpressionNoIn* are evaluated instead of the contained *ConditionalExpression* and *AssignmentExpression*, respectively.

11.13.1 Simple Assignment (=)

The production *AssignmentExpression* : *LeftHandSideExpression* **=** *AssignmentExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. Let *rref* be the result of evaluating *AssignmentExpression*.
3. Let *rval* be GetValue(*rref*).
4. 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"`**
5. Call PutValue(*lref*, *rval*).
6. Return *rval*.

NOTE When an assignment occurs within strict mode code, its *LeftHandSide* must not evaluate to an unresolvable reference. If it do es a **ReferenceError** exception is thrown upon assignment. 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 property has the value **false**. In these cases a **TypeError** exception is thrown.

11.13.2 Compound Assignment (op=)

The production *AssignmentExpression* : *LeftHandSideExpression AssignmentOperator AssignmentExpression*, where *AssignmentOperator* is @= and @ represents one of the operators indicated above, is evaluated as follows:

1. Let *lref* be the result of evaluating *LeftHandSideExpression*.
2. Let *lval* be GetValue(*lref*).
3. Let *rref* be the result of evaluating *AssignmentExpression*.
4. Let *rval* be GetValue(*rref*).
5. Let *r* be the result of applying operator @ to *lval* and *rval*.
6. 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"`**
7. Call PutValue(*lref*, *r*).
8. Return *r*.

NOTE See NOTE 11.13.1.

11.14 Comma Operator (,)

Syntax

Expression **:**
AssignmentExpression
Expression **,** *AssignmentExpression*

ExpressionNoIn **:**
AssignmentExpressionNoIn
ExpressionNoIn **,** *AssignmentExpressionNoIn*

Semantics

The production *Expression* : *Expression* **,** *AssignmentExpression* is evaluated as follows:

1. Let *lref* be the result of evaluating *Expression*.
2. Call GetValue(*lref*).

3. Let *rref* be the result of evaluating *AssignmentExpression*.
4. Return GetValue(*rref*).

The *ExpressionNoIn* production is evaluated in the same manner as the *Expression* production except that the contained *ExpressionNoIn* and *AssignmentExpressionNoIn* are evaluated instead of the contained *Expression* and *AssignmentExpression*, respectively.

NOTE GetValue must be called even though its value is not used because it may have observable side-effects.

12 Statements

Syntax

Statement **:**
Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement
LabelledStatement
SwitchStatement
ThrowStatement
TryStatement
DebuggerStatement

Semantics

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.

The result of evaluating a *Statement* is always a Completion value.

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 ECMAScript implementations either disallow this usage of *FunctionDeclaration* or issue a warning when such a usage is encountered. Future editions of ECMAScript may define alternative portable means for declaring functions in a *Statement* context.

12.1 Block

Syntax

Block **:**
{ $StatementList_{opt}$ **}**

StatementList **:**
Statement
StatementList Statement

Semantics

The production *Block* : **{ }** is evaluated as follows:

1. Return (**normal**, **empty**, **empty**).

The production *Block* : **{** *StatementList* **}** is evaluated as follows:

1. Return the result of evaluating *StatementList*.

The production *StatementList* : *Statement* is evaluated as follows:

1. Let *s* be the result of evaluating *Statement*.
2. If an exception was thrown, return (throw, *V*, empty) where *V* is the exception. (Execution now proceeds as if no exception were thrown.)
3. Return *s*.

The production *StatementList* : *StatementList Statement* is evaluated as follows:

1. Let *sl* be the result of evaluating *StatementList*.
2. If *sl* is an abrupt completion, return *sl*.
3. Let *s* be the result of evaluating *Statement*.
4. If an exception was thrown, return (throw, *V*, empty) where *V* is the exception. (Execution now proceeds as if no exception were thrown.)
5. If *s*.value is empty, let *V* = *sl*.value, otherwise let *V* = *s*.value.
6. Return (*s*.type, *V*, *s*.target).

NOTE Steps 5 and 6 of the above algoritm 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;")
```

12.2 Variable Statement

Syntax

VariableStatement :
var *VariableDeclarationList* **;**

VariableDeclarationList :
VariableDeclaration
VariableDeclarationList **,** *VariableDeclaration*

VariableDeclarationListNoIn :
VariableDeclarationNoIn
VariableDeclarationListNoIn **,** *VariableDeclarationNoIn*

VariableDeclaration :
Identifier $Initialiser_{opt}$

VariableDeclarationNoIn :
Identifier $InitialiserNoIn_{opt}$

Initialiser :
= *AssignmentExpression*

InitialiserNoIn :
= *AssignmentExpressionNoIn*

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 assigned 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 follows:

1. Evaluate *VariableDeclarationList*.
2. Return (normal, empty, empty).

The production *VariableDeclarationList* :*VariableDeclaration* is evaluated as follows:

1. Evaluate *VariableDeclaration*.

The production *VariableDeclarationList* : *VariableDeclarationList* **`,`** *VariableDeclaration* is evaluated as follows:

1. Evaluate *VariableDeclarationList*.
2. Evaluate *VariableDeclaration*.

The production *VariableDeclaration* : *Identifier* is evaluated as follows:

1. Return a String value containing the same sequence of characters as in the *Identifier*.

The production *VariableDeclaration* : *Identifier Initialiser* is evaluated as follows:

1. Let *lhs* be the result of evaluating *Identifier* as described in 11.1.2.
2. Let *rhs* be the result of evaluating *Initialiser*.
3. Let *value* be GetValue(*rhs*).
4. Call PutValue(*lhs*, *value*).
5. Return a String value containing the same sequence of characters as in the *Identifier*.

NOTE The String value of a *VariableDeclaration* is used in the evaluation of for-in statements (12.6.4).

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 4 will assign value to the property instead of to the VariableEnvironment binding of the *Identifier*.

The production *Initialiser* : **=** *AssignmentExpression* is evaluated as follows:

1. Return the result of evaluating *AssignmentExpression*.

The *VariableDeclarationListNoIn*, *VariableDeclarationNoIn* and *InitialiserNoIn* productions are evaluated in the same manner as the *VariableDeclarationList*, *VariableDeclaration* and *Initialiser* productions except that the contained *VariableDeclarationListNoIn*, *VariableDeclarationNoIn*, *InitialiserNoIn* and *AssignmentExpressionNoIn* are evaluated instead of the contained *VariableDeclarationList*, *VariableDeclaration*, *Initialiser* and *AssignmentExpression*, respectively.

12.2.1 Strict Mode Restrictions

It is a **SyntaxError** if a *VariableDeclaration* or *VariableDeclarationNoIn* occurs within strict code and its *Identifier* is either **`"eval"`** or **`"arguments"`**.

12.3 Empty Statement

Syntax

EmptyStatement :
`;`

Semantics

The production *EmptyStatement* : **;** is evaluated as follows:

1. Return (normal, empty, empty).

12.4 Expression Statement

Syntax

ExpressionStatement :
[lookahead ∉ {**{**, **function**}] *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** keyword because that might make it ambiguous with a *FunctionDeclaration*.

Semantics

The production *ExpressionStatement* : [lookahead ∉ {**{**, **function**}] *Expression* **;** is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return (normal, GetValue(*exprRef*), empty).

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**.

Semantics

The production *IfStatement* : **if (** *Expression* **)** *Statement* **else** *Statement* is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. If ToBoolean(GetValue(*exprRef*)) is **true**, then
   a. Return the result of evaluating the first *Statement*.
3. Else,
   a. Return the result of evaluating the second *Statement*.

The production *IfStatement* : **if (** *Expression* **)** *Statement* is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, empty, empty).
3. Return the result of evaluating *Statement*.

12.6 Iteration Statements

Syntax

IterationStatement :
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*

12.6.1 The do-while Statement

The production **do** *Statement* **while (** *Expression* **);** is evaluated as follows:

1. Let *V* = empty.
2. Let *iterating* be **true**.
3. Repeat, while *iterating* is **true**
    a. Let *stmt* be the result of evaluating *Statement*.
    b. If *stmt*.value is not empty, let *V* = *stmt*.value.
    c. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
        i. If *stmt*.type is break and *stmt*.target is in the current label set, return (normal, *V*, empty).
        ii. If *stmt* is an abrupt completion, return *stmt*.
    d. Let *exprRef* be the result of evaluating *Expression*.
    e. If ToBoolean(GetValue(*exprRef*)) is **false**, set *iterating* to **false**.
4. Return (normal, *V*, empty);

12.6.2 The while Statement

The production *IterationStatement* : **while (** *Expression* **)** *Statement* is evaluated as follows:

1. Let *V* = empty.
2. Repeat
    a. Let *exprRef* be the result of evaluating *Expression*.
    b. If ToBoolean(GetValue(*exprRef*)) is **false**, return (normal, *V*, empty).
    c. Let *stmt* be the result of evaluating *Statement*.
    d. If *stmt*.value is not empty, let *V* = *stmt*.value.
    e. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
        i. If *stmt*.type is break and *stmt*.target is in the current label set, then
            1. Return (normal, *V*, empty).
        ii. If *stmt* is an abrupt completion, return *stmt*.

12.6.3 The for Statement

The production
IterationStatement : **for (***ExpressionNoIn*$_{opt}$ **;** *Expression*$_{opt}$ **;** *Expression*$_{opt}$**)** *Statement*
is evaluated as follows:

1. If *ExpressionNoIn* is present, then.
    a. Let *exprRef* be the result of evaluating *ExpressionNoIn*.
    b. Call GetValue(*exprRef*). (This value is not used but the call may have side-effects.)
2. Let *V* = empty.
3. Repeat
    a. If the first *Expression* is present, then
        i. Let *testExprRef* be the result of evaluating the first *Expression*.
        ii. If ToBoolean(GetValue(*testExprRef*)) is **false**, return (normal, *V*, empty).
    b. Let *stmt* be the result of evaluating *Statement*.

c. If *stmt*.value is not empty, let *V* = *stmt*.value
d. If *stmt*.type is break and *stmt*.target is in the current label set, return (**normal**, *V*, **empty**).
e. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
    i. If *stmt* is an abrupt completion, return *stmt*.
f. If the second *Expression* is present, then
    i. Let *incExprRef* be the result of evaluating the second *Expression*.
    ii. Call GetValue(*incExprRef*). (This value is not used.)

The production
IterationStatement : **for (var** *VariableDeclarationListNoIn* **;** $Expression_{opt}$ **;** $Expression_{opt}$ **)** *Statement*
is evaluated as follows:

1. Evaluate *VariableDeclarationListNoIn*.
2. Let V = empty.
3. Repeat
    a. If the first *Expression* is present, then
        i. Let *testExprRef* be the result of evaluating the first *Expression*.
        ii. If ToBoolean(GetValue(*testExprRef*)) is **false**, then return (normal, V, empty).
    b. Let *stmt* be the result of evaluating *Statement*.
    c. If *stmt*.value is not empty, let V = *stmt*.value.
    d. If *stmt*.type is break and *stmt*.target is in the current label set, return (**normal**, V, **empty**).
    e. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
        i. If *stmt* is an abrupt completion, return *stmt*.
    f. If the second *Expression* is present, then.
        i. Let *incExprRef* be the result of evaluating the second *Expression*.
        ii. Call GetValue(*incExprRef*). (This value is not used.)

12.6.4 The `for-in` Statement

The production *IterationStatement* : **for (** *LeftHandSideExpression* **in** *Expression* **)** *Statement* is evaluated as follows:

1. Let *exprRef* be the result of evaluating the *Expression*.
2. Let *experValue* be GetValue(*exprRef*).
3. If *experValue* is **null** or **undefined**, return (normal, empty, empty).
4. Let *obj* be ToObject(*experValue*).
5. Let *V* = empty.
6. Repeat
    a. Let *P* be the name of the next property of *obj* whose [[Enumerable]] attribute is **true**. If there is no such property, return (normal, *V*, empty).
    b. Let *lhsRef* be the result of evaluating the *LeftHandSideExpression* (it may be evaluated repeatedly).
    c. Call PutValue(*lhsRef*, *P*).
    d. Let *stmt* be the result of evaluating *Statement*.
    e. If *stmt*.value is not empty, let *V* = *stmt*.value.
    f. If *stmt*.type is break and *stmt*.target is in the current label set, return (normal, *V*, empty).
    g. If *stmt*.type is not **continue** || *stmt*.target is not in the current label set, then
        i. If *stmt* is an abrupt completion, return *stmt*.

The production
IterationStatement : **for (var** *VariableDeclarationNoIn* **in** *Expression* **)** *Statement*
is evaluated as follows:

1. Let *varName* be the result of evaluating *VariableDeclarationNoIn*.
2. Let *exprRef* be the result of evaluating the *Expression*.
3. Let *experValue* be GetValue(*exprRef*).
4. If *experValue* is **null** or **undefined**, return (normal, empty, empty).
5. Let *obj* be ToObject(*experValue*).
6. Let *V* = empty.

7. Repeat
    a. Let *P* be the name of the next property of *obj* whose [[Enumerable]] attribute is **true**. If there is no such property, return (normal, *V*, empty).
    b. Let *varRef* be the result of evaluating *varName* as if it were an Identifier Reference (11.1.2); it may be evaluated repeatedly.
    c. Call PutValue(*varRef*, *P*).
    d. Let *stmt* be the result of evaluating *Statement*.
    e. If *stmt*.value is not empty, let *V* = *stmt*.value.
    f. If *stmt*.type is break and *stmt*.target is in the current label set, return (normal, *V*, empty).
    g. If *stmt*.type is not continue || *stmt*.target is not in the current label set, then
        i. If *stmt* is an abrupt completion, return *stmt*.

The mechanics and order of enumerating the properties (step 6.a in the first algorithm, step 7.a in the second) is not specified. 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.

NOTE See NOTE 11.13.1.

12.7 The `continue` Statement

Syntax

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

Semantics

A program is considered syntactically incorrect if either of the following is true:

- The program contains a **`continue`** statement without the optional *Identifier*, which is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement*.
- The program contains a **`continue`** statement with the optional *Identifier*, where *Identifier* does not appear in the label set of an enclosing (but not crossing function boundaries) *IterationStatement*.

A *ContinueStatement* without an *Identifier* is evaluated as follows:

1. Return (continue, empty, empty).

A *ContinueStatement* with the optional *Identifier* is evaluated as follows:

1. Return (continue, empty, *Identifier*).

12.8 The `break` Statement

Syntax

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

Semantics

A program is considered syntactically incorrect if either of the following is true:

- The program contains a **`break`** statement without the optional *Identifier*, which is not nested, directly or indirectly (but not crossing function boundaries), within an *IterationStatement* or a *SwitchStatement*.
- The program contains a **`break`** statement with the optional *Identifier*, where *Identifier* does not appear in the label set of an enclosing (but not crossing function boundaries) *Statement*.

A *BreakStatement* without an *Identifier* is evaluated as follows:

1. Return (break, empty, empty).

A *BreakStatement* with an *Identifier* is evaluated as follows:

1. Return (break, empty, *Identifier*).

12.9 The `return` Statement

Syntax

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

Semantics

An ECMAScript program is considered syntactically incorrect if it contains a **`return`** statement that is not within a *FunctionBody*. 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*.

A *ReturnStatement* is evaluated as follows:

1. If the *Expression* is not present, return (return, undefined, empty).
2. Let *exprRef* be the result of evaluating *Expression*.
3. Return (return, GetValue(*exprRef*), empty).

12.10 The `with` Statement

Syntax

WithStatement :
`with` **`(`** *Expression* **`)`** *Statement*

The **`with`** statement adds an object environment record for a computed object to the lexical environment of the current execution context. It then executes a statement using this augmented lexical environment. Finally, it restores the original lexical environment.

Semantics

The production *WithStatement* : **`with`** **(** *Expression* **)** *Statement* is evaluated as follows:

1. Let *val* be the result of evaluating *Expression*.
2. Let *obj* be ToObject(GetValue(*val*)).
3. Let *oldEnv* be the running execution context's LexicalEnvironment.
4. Let *newEnv* be the result of calling NewObjectEnvironment passing *obj* and *oldEnv* as the arguments.
5. Set the *provideThis* flag of *newEnv* to **true**.
6. Set the running execution context's LexicalEnvironment to *newEnv*.
7. Let *C* be the result of evaluating *Statement* but if an exception is thrown during the evaluation, let *C* be (throw, *V*, empty), where *V* is the exception. (Execution now proceeds as if no exception were thrown.)
8. Set the running execution context's Lexical Environment to *oldEnv*.
9. 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.

12.10.1 Strict Mode Restrictions

Strict mode code may not include a *WithStatement*. The occurrence of a *WithStatement* in such a context is treated as a **SyntaxError**.

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` **:** $StatementList_{opt}$

Semantics

The production *SwitchStatement* : **`switch`** **(** *Expression* **)** *CaseBlock* is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. Let *R* be the result of evaluating *CaseBlock*, passing it GetValue(*exprRef*) as a parameter.
3. If *R*.type is break and *R*.target is in the current label set, return (normal, *R*.value, empty).
4. Return *R*.

The production *CaseBlock* : { $CaseClauses_{opt}$ } is given an input parameter, *input*, and is evaluated as follows:

1. Let *V* = empty.
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 (normal, *V*, empty).

b. Let *clauseSelector* be the result of evaluating *C*.
c. If *input* is equal to *clauseSelector* as defined by the === operator, 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. If *R* is an abrupt completion, then return *R*.
3. Let *V* = *R*.value.

5. Repeat
a. Let *C* be the next *CaseClause* in *A*. If there is no such *CaseClause*, return (normal, *V*, empty).
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 (*R*.type, *V*, *R*.target).

The production *CaseBlock* : { *CaseClauses*$_{opt}$ *DefaultClause* *CaseClauses*$_{opt}$ } is given an input parameter, *input*, and is evaluated as follows:

1. Let *V* = empty.
2. Let *A* be the list of *CaseClause* items in the first *CaseClauses*, in source text order.
3. Let *B* be the list of *CaseClause* items in the second *CaseClauses*, in source text order.
4. Let *found* be **false**.
5. Repeat letting *C* be in order each *CaseClause* in *A*
a. If *found* is **false**, then
i. Let *clauseSelector* be the result of evaluating *C*.
ii. If *input* is equal to *clauseSelector* as defined by the === operator, then set *found* to **true**.
b. If *found* is **true**, then
i. If *C* has a *StatementList*, then
1. Evaluate *C*'s *StatementList* and let *R* be the result.
2. If *R*.value is not empty, then let *V* = *R*.value.
3. *R* is an abrupt completion, then return (*R*.type, *V*, *R*.target).
6. Let *foundInB* be **false**.
7. If *found* is **false**, then
a. Repeat, while *foundInB* is **false** and all elements of *B* have not been processed
i. Let *C* be the next *CaseClause* in *B*.
ii. Let *clauseSelector* be the result of evaluating *C*.
iii. If *input* is equal to *clauseSelector* as defined by the === operator, then
1. Set *foundInB* to **true**.
2. If *C* has a *StatementList*, then
a Evaluate *C*'s *StatementList* and let *R* be the result.
b If *R*.value is not empty, then let *V* = *R*.value.
c *R* is an abrupt completion, then return (*R*.type, *V*, *R*.target).
8. If *foundInB* is **false** and the *DefaultClause* has a *StatementList*, then
a. Evaluate the *DefaultClause's StatementList* and let *R* be the result.
b. If *R*.value is not empty, then let *V* = *R*.value.
c. If *R* is an abrupt completion, then return (*R*.type, *V*, *R*.target).
9. Repeat (Note that if step 7.a.i has been performed this loop does not start at the beginning of *B*)
a. Let *C* be the next *CaseClause* in *B*. If there is no such *CaseClause*, return (normal, *V*, empty).
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 (*R*.type, *V*, *R*.target).

The production *CaseClause* : **case** *Expression* : *StatementList*$_{opt}$ is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return GetValue(*exprRef*).

NOTE Evaluating *CaseClause* 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.

12.12 Labelled Statements

Syntax

LabelledStatement **:**
Identifier **:** *Statement*

Semantics

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.

An ECMAScript program is considered syntactically incorrect if it contains a *LabelledStatement* that is enclosed by a *LabelledStatement* with the same *Identifier* as label. This does not apply to labels appearing within the body of a *FunctionDeclaration* that is nested, directly or indirectly, within a labelled statement.

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**).

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**.

12.13 The `throw` Statement

Syntax

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

Semantics

The production *ThrowStatement* **:** **`throw`** [no *LineTerminator* here] *Expression* **`;`** is evaluated as follows:

1. Let *exprRef* be the result of evaluating *Expression*.
2. Return (throw, GetValue(*exprRef*), empty).

12.14 The `try` Statement

Syntax

TryStatement **:**
`try` *Block Catch*
`try` *Block Finally*
`try` *Block Catch Finally*

Catch **:**
`catch (` *Identifier* **`)`** *Block*

Finally **:**
`finally` *Block*

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 *Identifier* is bound to that exception.

Semantics

The production *TryStatement* : **`try`** *Block Catch* is evaluated as follows:

1. Let *B* be the result of evaluating *Block*.
2. If *B*.type is not throw, return *B*.
3. Return the result of evaluating *Catch* with parameter *B*.value.

The production *TryStatement* : **`try`** *Block Finally* is evaluated as follows:

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*.

The production *TryStatement* : **`try`** *Block Catch Finally* is evaluated as follows:

1. Let *B* be the result of evaluating *Block*.
2. If *B*.type is throw, then
    a. Let *C* be the result of evaluating *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*.

The production *Catch* : **`catch (`** *Identifier* **`)`** *Block* is evaluated as follows:

1. Let *C* be the parameter that has been passed to this production.
2. Let *oldEnv* be the running execution context's LexicalEnvironment.
3. Let *catchEnv* be the result of calling NewDeclarativeEnvironment passing *oldEnv* as the argument.
4. Call the CreateMutableBinding concrete method of *catchEnv* passing the *Identifier* String value as the argument.
5. Call the SetMutableBinding concrete method of *catchEnv* passing the *Identifier*, *C*, and **false** as arguments. Note that the last argument is immaterial in this situation.
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.

The production *Finally* : **`finally`** *Block* is evaluated as follows:

1. Return the result of evaluating *Block*.

12.14.1 Strict Mode Restrictions

It is a **SyntaxError** if a *TryStatement* with a *Catch* occurs within strict code and the *Identifier* of the *Catch* production is either **`"eval"`** or **`"arguments"`**.

12.15 The `debugger` statement

Syntax

DebuggerStatement :
`debugger ;`

Semantics

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 (normal, empty, empty).
3. Return *result*.

13 Function Definition

Syntax

FunctionDeclaration :
function *Identifier* **(** *FormalParameterList*$_{opt}$ **)** **{** *FunctionBody* **}**

FunctionExpression :
function *Identifier*$_{opt}$ **(** *FormalParameterList*$_{opt}$ **)** **{** *FunctionBody* **}**

FormalParameterList :
Identifier
FormalParameterList **,** *Identifier*

FunctionBody :
SourceElements$_{opt}$

Semantics

The production
FunctionDeclaration : **function** *Identifier* **(** *FormalParameterList*$_{opt}$ **)** **{** *FunctionBody* **}**
is instantiated as follows during Declaration Binding instantiation (10.5):

1. Return the result of creating a new Function object as specified in 13.2 with parameters specified by *FormalParameterList*$_{opt}$, and body specified by *FunctionBody*. 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.

The production
FunctionExpression : **function (** *FormalParameterList*$_{opt}$ **)** **{** *FunctionBody* **}**
is evaluated as follows:

1. Return the result of creating a new Function object as specified in 13.2 with parameters specified by *FormalParameterList*$_{opt}$ and body specified by *FunctionBody*. 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.

The production
FunctionExpression : **function** *Identifier* **(** *FormalParameterList*$_{opt}$ **)** **{** *FunctionBody* **}**
is evaluated as follows:

1. Let *funcEnv* be the result of calling NewDeclarativeEnvironment passing the running execution context's Lexical Environment as the argument
2. Let *envRec* be *funcEnv's* environment record.
3. Call the CreateImmutableBinding concrete method of *envRec* passing the String value of *Identifier* as the argument.

4. Let *closure* be the result of creating a new Function object as specified in 13.2 with parameters specified by $FormalParameterList_{opt}$ and body specified by *FunctionBody*. Pass in *funcEnv* 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.
5. Call the InitializeImmutableBinding concrete method of *envRec* passing the String value of *Identifier* and *closure* as the arguments.
6. Return *closure*.

NOTE The Identifier 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 *Identifier* in a *FunctionExpression* cannot be referenced from and does not affect the scope enclosing the *FunctionExpression*.

The production *FunctionBody* : $SourceElements_{opt}$ is evaluated as follows:

1. The code of this *FunctionBody* is strict mode code if it is part of a *FunctionDeclaration* or *FunctionExpression* that is contained in strict mode code or if the Directive Prologue (14.1) of its *SourceElements* 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, *SourceElements* is evaluated in the following steps as strict mode code. Otherwise, *SourceElements* is evaluated in the following steps as non-strict mode code.
2. If *SourceElements* is present return the result of evaluating *SourceElements*.
3. Else return (normal, **undefined**, empty).

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*.

13.2 Creating Function Objects

Given an optional parameter list specified by *FormalParameterList*, a body specified by *FunctionBody*, a Lexical Environment specified by *Scope*, and a Boolean flag *Strict*, a Function object is constructed as follows:

1. Create a new native ECMAScript object and let *F* be that object.
2. Set all the internal methods, except for [[Get]], of *F* as described in 8.12.
3. Set the [[Class]] internal property of *F* to **"Function"**.
4. Set the [[Prototype]] internal property of *F* to the standard built-in Function prototype object as specified in 15.3.3.1.
5. Set the [[Get]] internal property of *F* as described in 15.3.5.4.
6. Set the [[Call]] internal property of *F* as described in 13.2.1.
7. Set the [[Construct]] internal property of *F* as described in 13.2.2.
8. Set the [[HasInstance]] internal property of *F* as described in 15.3.5.3.
9. Set the [[Scope]] internal property of *F* to the value of *Scope*.
10. Let *names* be a List containing, in left to right textual order, the Strings corresponding to the identifiers of *FormalParameterList*. If no parameters are specified, let *names* be the empty list.
11. Set the [[FormalParameters]] internal property of *F* to *names*.
12. Set the [[Code]] internal property of *F* to *FunctionBody*.
13. Set the [[Extensible]] internal property of *F* to **true**.
14. Let *len* be the number of formal parameters specified in *FormalParameterList*. If no parameters are specified, let *len* be 0.
15. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"length"**, Property Descriptor {[[Value]]: *len*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
16. Let *proto* be the result of creating a new object as would be constructed by the expression **new Object()** where **Object** is the standard built-in constructor with that name.
17. Call the [[DefineOwnProperty]] internal method of *proto* with arguments **"constructor"**, Property Descriptor {[[Value]]: *F*, { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true**}, and **false**.

18. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"prototype"**, Property Descriptor {[[Value]]: *proto*, { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
19. If *Strict* is **true**, then
    a. Let *thrower* be the [[ThrowTypeError]] function Object (13.2.3).
    b. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"caller"**, PropertyDescriptor {[[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
    c. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"arguments"**, PropertyDescriptor {[[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
20. Return *F*.

NOTE A **prototype** property is automatically created for every function, to allow for the possibility that the function will be used as a constructor.

13.2.1 [[Call]]

When the [[Call]] internal method for a Function object *F* is called with a this value and a list of arguments, the following steps are taken:

1. Let *funcCtx* be the result of establishing a new execution context for function code using the value of *F*'s [[FormalParameters]] internal property, the passed arguments List *args*, and the **this** value as described in 10.4.3.
2. Let *result* be the result of evaluating the *FunctionBody* that is the value of *F*'s [[Code]] internal property. If *F* does not have a [[Code]] internal property or if its value is an empty *FunctionBody*, then *result* is (normal, undefined, empty).
3. Exit the execution context *funcCtx*, restoring the previous execution context.
4. If *result*.type is throw then throw *result*.value.
5. If *result*.type is return then return *result*.value.
6. Otherwise *result*.type must be normal. Return **undefined**.

13.2.2 [[Construct]]

When the [[Construct]] internal method for a Function object *F* is called with a possibly empty list of arguments, the following steps are taken:

1. Let *obj* be a newly created native ECMAScript object.
2. Set all the internal methods of *obj* as specified in 8.12.
3. Set the [[Class]] internal property of *obj* to **"Object"**.
4. Set the [[Extensible]] internal property of *obj* to **true**.
5. Let *proto* be the value of calling the [[Get]] internal property of *F* with argument **"prototype"**.
6. If Type(*proto*) is Object, set the [[Prototype]] internal property of *obj* to *proto*.
7. If Type(*proto*) is not Object, set the [[Prototype]] internal property of *obj* to the standard built-in Object prototype object as described in 15.2.4.
8. Let *result* be the result of calling the [[Call]] internal property of *F*, providing *obj* as the **this** value and providing the argument list passed into [[Construct]] as *args*.
9. If Type(*result*) is Object then return *result*.
10. Return *obj*.

13.2.3 The [[ThrowTypeError]] Function Object

The [[ThrowTypeError]] object is a unique function object that is defined once as follows:

1. Create a new native ECMAScript object and let *F* be that object.
2. Set all the internal methods of *F* as described in 8.12.
3. Set the [[Class]] internal property of *F* to **"Function"**.
4. Set the [[Prototype]] internal property of *F* to the standard built-in Function prototype object as specified in 15.3.3.1.
5. Set the [[Call]] internal property of *F* as described in 13.2.1.
6. Set the [[Scope]] internal property of *F* to the Global Environment.
7. Set the [[FormalParameters]] internal property of *F* to an empty List.

8. Set the [[Code]] internal property of *F* to be a *FunctionBody* that unconditionally throws a **TypeError** exception and performs no other action.
9. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"length"**, Property Descriptor {[[Value]]: *0*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
10. Set the [[Extensible]] internal property of *F* to **false**.
11. Let [[ThrowTypeError]] be *F*.

14 Program

Syntax

Program **:**
SourceElements$_{opt}$

SourceElements **:**
SourceElement
SourceElements SourceElement

SourceElement **:**
Statement
FunctionDeclaration

Semantics

The production *Program* **:** *SourceElements*$_{opt}$ is evaluated as follows:

1. The code of this *Program* is strict mode code if the Directive Prologue (14.1) of its *SourceElements* contains a Use Strict Directive or if any of the conditions of 10.1.1 apply. If the code of this *Program* is strict mode code, *SourceElements* is evaluated in the following steps as strict mode code. Otherwise *SourceElements* is evaluated in the following steps as non-strict mode code.
2. If *SourceElements* is not present, return (normal, empty, empty).
3. Let *progCxt* be a new execution context for global code as described in 10.4.1.
4. Let *result* be the result of evaluating *SourceElements*.
5. Exit the execution context *progCxt*.
6. Return *result*.

NOTE The processes for initiating the evaluation of a *Program* and for dealing with the result of such an evaluation are defined by an ECMAScript implementation and not by this specification.

The production *SourceElements* **:** *SourceElements SourceElement* is evaluated as follows:

1. Let *headResult* be the result of evaluating *SourceElements*.
2. If *headResult* is an abrupt completion, return *headResult*.
3. Let *tailResult* be result of evaluating *SourceElement*.
4. If *tailResult*.value is empty, let *V* = *headResult*.value, otherwise let *V* = *tailResult*.value.
5. Return (tailResult.type, V, tailResult.target)

The production *SourceElement* **:** *Statement* is evaluated as follows:

1. Return the result of evaluating *Statement*.

The production *SourceElement* **:** *FunctionDeclaration* is evaluated as follows:

1. Return (normal, empty, empty).

14.1 Directive Prologues and the Use Strict Directive

A Directive Prologue is the longest sequence of *ExpressionStatement* productions occurring as the initial *SourceElement* productions of a *Program* or *FunctionBody* and where each *ExpressionStatement* in the sequence

consists entirely of a *StringLiteral* token followed 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 *SourceElements* 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.

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, the [[Class]] internal property of a built-in object is **"Function"** if that built-in object has a [[Call]] internal property, or **"Object"** if that built-in object does not have a [[Call]] internal property. Unless specified otherwise, the [[Extensible]] internal 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** (15.3.4), as the value of its [[Prototype]] internal 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 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. 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 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** }. Every other property described in this clause has the attributes { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **true** } unless otherwise specified.

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 property; it is not possible to use the global object as a constructor with the **new** operator.

The global object does not have a [[Call]] internal property; it is not possible to invoke the global object as a function.

The values of the [[Prototype]] and [[Class]] internal properties of the global object are implementation-dependent.

In addition to the properties defined in this specification the global object may have additional host defined properties. This may include a property whose value is the global object itself; for example, in the HTML document object model the **window** property of the global object is the global object itself.

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 *prog* be the ECMAScript code that is the result of parsing *x* as a *Program*. If the parse fails, throw a **SyntaxError** exception (but see also clause 16).
3. Let *evalCtx* be the result of establishing a new execution context (10.4.2) for the eval code *prog*.
4. Let *result* be the result of evaluating the program *prog*.
5. Exit the running execution context *evalCtx*, restoring the previous execution context.
6. If *result*.type is normal and its completion value is a value *V*, then return the value *V*.
7. If *result*.type is normal and its completion value is empty, then return the value **undefined**.
8. Otherwise, *result*.type must be throw. Throw *result*.value as an exception.

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. 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.
3. Let *sign* be 1.
4. If *S* is not empty and the first character of *S* is a minus sign -, let *sign* be −1.
5. 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*.
6. Let *R* = ToInt32(*radix*).
7. Let *stripPrefix* be **true**.
8. If *R* ≠ 0, then
    a. If *R* < 2 or *R* > 36, then return **NaN**.
    b. If *R* ≠ 16, let *stripPrefix* be **false**.
9. Else, *R* = 0
    a. Let *R* = 10.
10. 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.

11. 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*.
12. If *Z* is empty, return **NaN**.
13. 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.)
14. Let *number* be the Number value for *mathInt*.
15. Return *sign* × *number*.

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. 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.
3. If neither *trimmedString* nor any prefix of *trimmedString* satisfies the syntax of a *StrDecimalLiteral* (see 9.3.1), return **NaN**.
4. Let *numberString* be the longest prefix of *trimmedString*, which might be *trimmedString* itself, that satisfies the syntax of a *StrDecimalLiteral*.
5. 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. If ToNumber(*number*) is **NaN**, return **true**.
2. Otherwise, return **false**.

NOTE A reliable way for ECMAScript code to test if a value `X` is a **NaN** is an expression of the form `X !== X`. The result will be **true** if and only if `X` is a **NaN**.

15.1.2.5 isFinite (number)

Returns **false** if the argument coerces to **NaN**, **+∞**, or **–∞**, and otherwise returns **true**.

1. If ToNumber(*number*) is **NaN**, +∞, or –∞, return **false**.
2. Otherwise, return **true**.

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.

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".

The encoding and es caping 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.
        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.

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* is not '%', throw a **URIError** exception.
    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* is less than 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* is ≥ 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 21.

Table 21 — 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.

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. Let *reservedURISet* be a String containing one instance of each character valid in *uriReserved* plus "#".
3. 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 decodeURIComponent (encodedURIComponent)

The **decodeURIComponent** function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the **encodeURIComponent** function is replaced with the 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. Let *reservedURIComponentSet* be the empty String.
3. 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. Let *unescapedURISet* be a String containing one instance of each character valid in *uriReserved* and *uriUnescaped* plus "#".
3. 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. Let *unescapedURIComponentSet* be a String containing one instance of each character valid in *uriUnescaped*.
3. 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 and 15.3.2.

15.1.4.3 Array (. . .)

See 15.4.1 and 15.4.2.

15.1.4.4 String (. . .)

See 15.5.1 and 15.5.2.

15.1.4.5 Boolean (. . .)

See 15.6.1 and 15.6.2.

15.1.4.6 Number (. . .)

See 15.7.1 and 15.7.2.

15.1.4.7 Date (. . .)

See 15.9.2.

15.1.4.8 RegExp (. . .)

See 15.10.3 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 URIError (. . .)

See 15.11.6.6.

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, create and return a new Object object exactly as if the standard built-in Object constructor had been called with the same arguments (15.2.2.1).
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
        i. If the *value* is a native ECMAScript object, do not create a new object but simply return *value*.
        ii. If the *value* is a host object, then actions are taken and a result is returned in an implementation-dependent manner that may depend on the host object.
    b. If Type(*value*) is String, return ToObject(*value*).
    c. If Type(*value*) is Boolean, return ToObject(*value*).
    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. Let *obj* be a newly created native ECMAScript object.
4. Set the [[Prototype]] internal property of *obj* to the standard built-in Object prototype object (15.2.4).
5. Set the [[Class]] internal property of *obj* to **`"Object"`**.
6. Set the [[Extensible]] internal property of *obj* to **true**.
7. Set all the internal methods of *obj* as specified in 8.12.
8. Return *obj*.

15.2.3 Properties of the Object Constructor

The value of the [[Prototype]] internal property of the Object constructor is the standard built-in Function prototype object.

Besides the internal properties and 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. If Type(*O*) is not Object throw a **TypeError** exception.
2. Return the value of the [[Prototype]] internal property of *O*.

15.2.3.3 Object.getOwnPropertyDescriptor (O, P)

When the **getOwnPropertyDescriptor** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Let *name* be ToString(*P*).
3. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with argument *name*.
4. Return the result of calling FromPropertyDescriptor(*desc*) (8.10.4).

15.2.3.4 Object.getOwnPropertyNames (O)

When the **getOwnPropertyNames** function is called, the following steps are taken:

1. If Type(*O*) is not Object throw a **TypeError** exception.
2. Let *array* be the result of creating a new object as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.
3. Let *n* be 0.
4. For each named own property *P* of *O*
    a. Let *name* be the String value that is the name of *P*.
    b. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(*n*), the PropertyDescriptor {[[Value]]: *name*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    c. Increment *n* by 1.
5. Return *array*.

NOTE If *O* is a String instance, the set of own properties processed in step 4 includes the implicit properties defined in 15.5.5.2 that correspond to character positions within the object's [[PrimitiveValue]] String.

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 creating a new object as if by the expression new Object() where Object is the standard built-in constructor with that name
3. Set the [[Prototype]] internal property of *obj* to *O*.
4. If the argument *Properties* is present and not **undefined**, add own properties to *obj* as if by calling the standard built-in function `Object.defineProperties` with arguments *obj* and *Properties*.
5. 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 ToString(*P*).
3. Let *desc* be the result of calling ToPropertyDescriptor with *Attributes* as the argument.
4. Call the [[DefineOwnProperty]] internal method of *O* with arguments *name*, *desc*, and **true**.
5. 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. 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 names 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 calling the [[Get]] internal method of *props* with *P* as the argument.
    b. Let *desc* be the result of calling ToPropertyDescriptor with *descObj* as the argument.
    c. Append the pair (a two element List) consisting of *P* and *desc* to the end of *descriptors*.
6. 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. Call the [[DefineOwnProperty]] internal method of *O* with arguments *P*, *desc*, and **true**.
7. 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.

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 throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
    a. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
    b. If *desc*.[[Configurable]] is **true**, set *desc*.[[Configurable]] to **false**.
    c. Call the [[DefineOwnProperty]] internal method of *O* with *P*, *desc*, and **true** as arguments.
3. Set the [[Extensible]] internal property of *O* to **false**.
4. 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 a **TypeError** exception.
2. For each named own property name *P* of *O*,
    a. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
    b. If IsDataDescriptor(*desc*) is **true**, then
        i. If *desc*.[[Writable]] is **true**, set *desc*.[[Writable]] to **false**.
    c. If *desc*.[[Configurable]] is **true**, set *desc*.[[Configurable]] to **false**.
    d. Call the [[DefineOwnProperty]] internal method of *O* with *P*, *desc*, and **true** as arguments.
3. Set the [[Extensible]] internal property of *O* to **false**.
4. 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 a **TypeError** exception.
2. Set the [[Extensible]] internal property of *O* to **false**.
3. 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 throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
    a. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
    b. If *desc*.[[Configurable]] is **true**, then return **false**.
3. If the [[Extensible]] internal property of *O* is **false**, then return **true**.
4. Otherwise, return **false**.

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 throw a **TypeError** exception.
2. For each named own property name *P* of *O*,
    a. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* with *P*.
    b. If IsDataDescriptor(*desc*) is **true** then
        i. If *desc*.[[Writable]] is **true**, return **false**.
    c. If *desc*.[[Configurable]] is **true**, then return **false**.
3. If the [[Extensible]] internal property of *O* is **false**, then return **true**.
4. Otherwise, return **false**.

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 throw a **TypeError** exception.
2. Return the Boolean value of the [[Extensible]] internal property of *O*.

15.2.3.14 Object.keys (O)

When the **keys** function is called with argument *O*, the following steps are taken:

1. If the Type(*O*) is not Object, throw a **TypeError** exception.
2. Let *n* be the number of own enumerable properties of *O*.
3. Let *array* be the result of creating a new Object as if by the expression **`new Array(n)`** where **`Array`** is the standard built-in constructor with that name.
4. Let *index* be 0.
5. For each own enumerable property of *O* whose name String is *P*
    a. Call the [[DefineOwnProperty]] internal method of *array* with arguments ToString(*index*), the PropertyDescriptor {[[Value]]: *P*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    b. Increment *index* by 1.
6. Return *array*.

If an implementation defines a specific order of enumeration for the for-in statement, that same enumeration order must be used in step 5 of this algorithm.

15.2.4 Properties of the Object Prototype Object

The value of the [[Prototype]] internal property of the Object prototype object is **null**, the value of the [[Class]] internal property is **`"Object"`**, and the initial value of the [[Extensible]] internal 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. Let *class* be the value of the [[Class]] internal property of *O*.
5. Return the String value that is the result of concatenating the three Strings **`"[object "`**, *class*, and **`"]"`**.

15.2.4.3 Object.prototype.toLocaleString ( )

When the **toLocaleString** 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. Let *toString* be the result of calling the [[Get]] internal method of *O* passing **"toString"** as the argument.
3. If IsCallable(*toString*) is **false**, throw a **TypeError** exception.
4. Return the result of calling the [[Call]] internal method of *toString* passing *O* as the **this** value and no arguments.

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. If *O* is the result of calling the Object constructor with a host object (15.2.2.1), then
    a. Return either *O* or another value such as the host object originally passed to the constructor. The specific result that is returned is implementation-defined.
3. 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 ToString(*V*).
2. Let *O* be the result of calling ToObject passing the **this** value as the argument.
3. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* passing *P* as the argument.
4. If *desc* is **undefined**, return **false**.
5. Return **true**.

NOTE 1 Unlike [[HasProperty]] (8.12.6), 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**.

15.2.4.6 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. Repeat
    a. Let *V* be the value of the [[Prototype]] internal property of *V*.
    b. if *V* is **null**, return **false**
    c. If *O* and *V* refer to the same object, return **true**.

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. Let *O* be the result of calling ToObject passing the **this** value as the argument.
3. Let *desc* be the result of calling the [[GetOwnProperty]] internal method of *O* passing *P* as the argument.
4. If *desc* is **undefined**, return **false**.
5. 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 Called as a Function

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.

15.3.1.1 Function (p1, p2, … , pn, body)

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. Create and return a new Function object as if the standard built-in constructor Function was used 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)

The last argument specifies the body (executable code) of a function; any preceding arguments specify formal parameters.

When the **`Function`** constructor 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 *body* be the empty String.
4. Else if *argCount* = 1, let *body* be that argument.
5. Else, *argCount* > 1
    a. Let *firstArg* be the first argument.
    b. Let *P* be ToString(*firstArg*).
    c. Let *k* be 2.
    d. Repeat, while *k* < *argCount*
        i. Let *nextArg* be the *k*'th argument.
        ii. Let *P* be the result of concatenating the previous value of *P*, the String **`","`** (a comma), and ToString(*nextArg*).
        iii. Increase *k* by 1.

e. Let *body* be the *k*'th argument.
6. Let *body* be ToString(*body*).
7. If *P* is not parsable as a *FormalParameterList*$_{opt}$ then throw a **SyntaxError** exception.
8. If *body* is not parsable as *FunctionBody* then throw a **SyntaxError** exception.
9. If *body* is strict mode code (see 10.1.1) then let *strict* be **true**, else let *strict* be **false**.
10. If *strict* is **true**, throw any exceptions specified in 13.1 that apply.
11. Return a new Function object created as specified in 13.2 passing *P* as the *FormalParameterList*$_{opt}$ and *body* as the *FunctionBody*. Pass in the Global Environment as the *Scope* parameter and *strict* as the *Strict* flag.

A **`prototype`** property is automatically created for every function, 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")
```

15.3.3 Properties of the Function Constructor

The Function constructor is itself a Function object and its [[Class]] is **`"Function"`**. The value of the [[Prototype]] internal property of the Function constructor is the standard built-in Function prototype object (15.3.4).

The value of the [[Extensible]] internal property of the Function constructor is **true**.

The Function constructor has the following properties:

15.3.3.1 Function.prototype

The initial value of **`Function.prototype`** is the standard built-in Function prototype object (15.3.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.3.3.2 Function.length

This is a data property with a value of 1. This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.3.4 Properties of the Function Prototype Object

The Function prototype object is itself a Function object (its [[Class]] is **`"Function"`**) that, when invoked, accepts any arguments and returns **undefined**.

The value of the [[Prototype]] internal property of the Function prototype object is the standard built-in Object prototype object (15.2.4). The initial value of the [[Extensible]] internal property of the Function prototype object is **true**.

The Function prototype object does not have a **`valueOf`** property of its own; however, it inherits the **`valueOf`** property from the Object prototype Object.

The **`length`** property of the Function prototype object is **`0`**.

15.3.4.1 Function.prototype.constructor

The initial value of **`Function.prototype.constructor`** is the built-in **`Function`** constructor.

15.3.4.2 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.

15.3.4.3 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:

1. If IsCallable(*func*) is **false**, then throw a **TypeError** exception.
2. If *argArray* is **null** or **undefined**, then
    a. Return the result of calling the [[Call]] internal method of *func*, providing *thisArg* as the **this** value and an empty list of arguments.
3. If Type(*argArray*) is not Object, then throw a **TypeError** exception.
4. Let *len* be the result of calling the [[Get]] internal method of *argArray* with argument **`"length"`**.
5. Let *n* be ToUint32(*len*).
6. Let *argList* be an empty List.
7. Let *index* be 0.
8. Repeat while *index* < *n*
    a. Let *indexName* be ToString(*index*).
    b. Let *nextArg* be the result of calling the [[Get]] internal method of *argArray* with *indexName* as the argument.
    c. Append *nextArg* as the last element of *argList*.
    d. Set *index* to *index* + 1.
9. Return the result of calling the [[Call]] internal method of *func*, providing *thisArg* as the **this** value and *argList* as the list of arguments.

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 2, 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.

15.3.4.4 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 the **this** value and *argList* as the list of arguments.

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 2, 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.

15.3.4.5 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 of all of the argument values provided after *thisArg* (*arg1*, *arg2* etc), in order.
4. Let *F* be a new native ECMAScript object .
5. Set all the internal methods, except for [[Get]], of *F* as specified in 8.12.
6. Set the [[Get]] internal property of *F* as specified in 15.3.5.4.
7. Set the [[TargetFunction]] internal property of *F* to *Target*.
8. Set the [[BoundThis]] internal property of *F* to the value of *thisArg*.
9. Set the [[BoundArgs]] internal property of *F* to *A*.
10. Set the [[Class]] internal property of *F* to "**Function**".
11. Set the [[Prototype]] internal property of *F* to the standard built-in Function prototype object as specified in 15.3.3.1.
12. Set the [[Call]] internal property of *F* as described in 15.3.4.5.1.
13. Set the [[Construct]] internal property of *F* as described in 15.3.4.5.2.
14. Set the [[HasInstance]] internal property of *F* as described in 15.3.4.5.3.
15. If the [[Class]] internal property of *Target* is "**Function**", then
    a. Let *L* be the **length** property of *Target* minus the length of *A*.
    b. Set the **length** own property of *F* to either 0 or *L*, whichever is larger.
16. Else set the **length** own property of *F* to 0.
17. Set the attributes of the **length** own property of *F* to the values specified in 15.3.5.1.
18. Set the [[Extensible]] internal property of *F* to **true**.
19. Let *thrower* be the [[ThrowTypeError]] function Object (13.2.3).
20. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"caller"**, PropertyDescriptor {[[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
21. Call the [[DefineOwnProperty]] internal method of *F* with arguments **"arguments"**, PropertyDescriptor {[[Get]]: *thrower*, [[Set]]: *thrower*, [[Enumerable]]: **false**, [[Configurable]]: **false**}, and **false**.
22. Return *F*.

The **length** property of the **bind** method is **1**.

NOTE Function objects created using **Function.prototype.bind** do not have a **prototype** property or the [[Code]], [[FormalParameters]], and [[Scope]] internal properties.

15.3.4.5.1 [[Call]]

When the [[Call]] internal method of a function object, *F*, which was created using the bind function is called with a **this** value and a list of arguments *ExtraArgs*, the following steps are taken:

1. Let *boundArgs* be the value of *F's* [[BoundArgs]] internal property.
2. Let *boundThis* be the value of *F's* [[BoundThis]] internal property.
3. Let *target* be the value of *F's* [[TargetFunction]] internal 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 [[Call]] internal method of *target* providing *boundThis* as the **this** value and providing *args* as the arguments.

15.3.4.5.2 [[Construct]]

When the [[Construct]] internal method of a 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* [[TargetFunction]] internal property.
2. If *target* has no [[Construct]] internal method, a **TypeError** exception is thrown.

3. Let *boundArgs* be the value of *F's* [[BoundArgs]] internal 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.

15.3.4.5.3 [[HasInstance]] (V)

When the [[HasInstance]] internal method of a function object *F*, that was created using the bind function is called with argument *V*, the following steps are taken:

1. Let *target* be the value of *F's* [[TargetFunction]] internal property.
2. If *target* has no [[HasInstance]] internal method, a **TypeError** exception is thrown.
3. Return the result of calling the [[HasInstance]] internal method of *target* providing *V* as the argument.

15.3.5 Properties of Function Instances

In addition to the required internal properties, every function instance has a [[Call]] internal property and in most cases uses a different version of the [[Get]] internal property. Depending on how they are created (see 8.6.2, 13.2, 15, and 15.3.4.5), function instances may have a [[HasInstance]] internal property, a [[Scope]] internal property, a [[Construct]] internal property, a [[FormalParameters]] internal property, a [[Code]] internal property, a [[TargetFunction]] internal property, a [[BoundThis]] internal property, and a [[BoundArgs]] internal property.

The value of the [[Class]] internal property is **"Function"**.

Function instances that correspond to strict mode functions (13.2) and function instances created using the **Function.prototype.bind method** (15.3.4.5) 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.

15.3.5.1 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** }.

15.3.5.2 prototype

The value of the **`prototype`** property is used to initialise the [[Prototype]] internal property of a newly created object before the Function object is invoked as a constructor for that newly created object. This property has the attribute { [[Writable]]: **true**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

NOTE Function objects created using **`Function.prototype.bind`** do not have a **`prototype`** property.

15.3.5.3 [[HasInstance]] (V)

Assume *F* is a Function object.

When the [[HasInstance]] internal method of *F* is called with value *V*, the following steps are taken:

1. If *V* is not an object, return **false**.
2. Let *O* be the result of calling the [[Get]] internal method of *F* with property name **`"prototype"`**.
3. If Type(*O*) is not Object, throw a **TypeError** exception.
4. Repeat
    a. Let *V* be the value of the [[Prototype]] internal property of *V*.

b. If *V* is **null**, return **false**.
c. If *O* and *V* refer to the same object, return **true**.

NOTE Function objects created using **Function.prototype.bind** have a different implementation of [[HasInstance]] defined in 15.3.4.5.3.

15.3.5.4 [[Get]] (P)

Function objects use a variation of the [[Get]] internal method used for other native ECMAScript objects (8.12.3).

Assume *F* is a Function object. When the [[Get]] internal method of *F* is called with property name *P*, the following steps are taken:

1. Let *v* be the result of calling the default [[Get]] internal method (8.12.3) on *F* passing *P* as the property name argument.
2. If *P* is **"caller"** and *v* is a strict mode Function object, throw a **TypeError** exception.
3. Return *v*.

NOTE Function objects created using **Function.prototype.bind** use the default [[Get]] internal method.

15.4 Array Objects

Array objects give 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.

An object, *O*, is said to be *sparse* if the following algorithm returns **true**:

1. Let *len* be the result of calling the [[Get]] internal method of *O* with argument **"length"**.
2. For each integer *i* in the range $0 \le i <$ 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 Called as a Function

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.

15.4.1.1 Array ([item1 [, item2 [, …]]])

When the **Array** function is called the following steps are taken:

1. Create and return a new Array object exactly as if the standard built-in constructor **Array** was used in a **new** expression with the same arguments (15.4.2).

15.4.2 The Array Constructor

When **`Array`** is called as part of a **new** expression, it is a constructor: it initialises the newly created object.

15.4.2.1 new Array ([item0 [, item1 [, …]]])

This description applies if and only if the Array constructor is given no arguments or at least two arguments.

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **`Array.prototype`** (15.4.3.1).

The [[Class]] internal property of the newly constructed object is set to **`"Array"`**.

The [[Extensible]] internal property of the newly constructed object is set to **true**.

The **`length`** property of the newly constructed object is set to the number of arguments.

The **`0`** property of the newly constructed object is set to *item0* (if supplied); the **`1`** property of the newly constructed object is set to *item1* (if supplied); and, in general, for as many arguments as there are, the *k* property of the newly constructed object is set to argument *k*, where the first argument is considered to be argument number **`0`**. These properties all have the attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

15.4.2.2 new Array (len)

The [[Prototype]] internal property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of **`Array.prototype`** (15.4.3.1). The [[Class]] internal property of the newly constructed object is set to **`"Array"`**. The [[Extensible]] internal property of the newly constructed object is set to **true**.

If the argument *len* is a Number and ToUint32(*len*) is equal to *len*, then the **`length`** property of the newly constructed object is set to ToUint32(*len*). If the argument *len* is a Number and ToUint32(*len*) is not equal to *len*, a **RangeError** exception is thrown.

If the argument *len* is not a Number, then the **`length`** property of the newly constructed object is set to **`1`** and the **`0`** property of the newly constructed object is set to *len* with attributes {[[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}.

15.4.3 Properties of the Array Constructor

The value of the [[Prototype]] internal property of the Array constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **`length`** property (whose value is **1**), the Array constructor has the following properties:

15.4.3.1 Array.prototype

The initial value of **`Array.prototype`** is the Array prototype object (15.4.4).

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.4.3.2 Array.isArray (arg)

The isArray function takes one argument *arg*, and returns the Boolean value **true** if the argument is an object whose class internal property is **"Array"**; otherwise it returns **false**. The following steps are taken:

1. If Type(*arg*) is not Object, return **false**.
2. If the value of the [[Class]] internal property of *arg* is **"Array"**, then return **true**.
3. Return **false**.

15.4.4 Properties of the Array Prototype Object

The value of the [[Prototype]] internal property of the Array prototype object is the standard built-in Object prototype object (15.2.4).

The Array prototype object is itself an array; its [[Class]] is **"Array"**, and it has a **length** property (whose initial value is **+0**) and the special [[DefineOwnProperty]] internal method described in 15.4.5.1.

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** to be an object for which the value of the [[Class]] internal property is not **"Array"**.

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.

15.4.4.1 Array.prototype.constructor

The initial value of **Array.prototype.constructor** is the standard built-in **Array** constructor.

15.4.4.2 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. Let *func* be the result of calling the [[Get]] internal method of *array* with argument **"join"**.
3. If IsCallable(*func*) is **false**, then let *func* be the standard built-in method Object.prototype.toString (15.2.4.2).
4. Return the result of calling the [[Call]] internal method of *func* providing *array* as the **this** value and an empty arguments list.

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 a host object is implementation-dependent.

15.4.4.3 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. Let *arrayLen* be the result of calling the [[Get]] internal method of *array* with argument **"length"**.
3. Let *len* be ToUint32(*arrayLen*).
4. 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).
5. If *len* is zero, return the empty String.

6. Let *firstElement* be the result of calling the [[Get]] internal method of *array* with argument **"0"**.
7. If *firstElement* is **undefined** or **null**, then
    a. Let *R* be the empty String.
8. Else
    a. Let *elementObj* be ToObject(*firstElement*).
    b. Let *func* be the result of calling the [[Get]] internal method of *elementObj* with argument **"toLocaleString"**.
    c. If IsCallable(*func*) is **false**, throw a **TypeError** exception.
    d. Let *R* be the result of calling the [[Call]] internal method of *func* providing *elementObj* as the **this** value and an empty arguments list.
9. Let *k* be **1**.
10. Repeat, while *k* < *len*
    a. Let *S* be a String value produced by concatenating *R* and *separator*.
    b. Let *nextElement* be the result of calling the [[Get]] internal method of *array* with argument ToString(*k*).
    c. If *nextElement* is **undefined** or **null**, then
        i. Let *R* be the empty String.
    d. Else
        i. Let *elementObj* be ToObject(*nextElement*).
        ii. Let *func* be the result of calling the [[Get]] internal method of *elementObj* with argument **"toLocaleString"**.
        iii. If IsCallable(*func*) is **false**, throw a **TypeError** exception.
        iv. Let *R* be the result of calling the [[Call]] internal method of *func* providing *elementObj* as the **this** value and an empty arguments list.
    e. Let *R* be a String value produced by concatenating *S* and *R*.
    f. Increase *k* by 1.
11. 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 a host object is implementation-dependent.

15.4.4.4 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 a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
3. Let *n* be 0.
4. 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.
5. Repeat, while *items* is not empty
    a. Remove the first element from *items* and let *E* be the value of the element.
    b. If the value of the [[Class]] internal property of *E* is **"Array"**, then
        i. Let *k* be 0.
        ii. Let *len* be the result of calling the [[Get]] internal method of *E* with argument **"length"**.
        iii. Repeat, while *k* < *len*
            1. Let *P* be ToString(*k*).
            2. Let *exists* be the result of calling the [[HasProperty]] internal method of *E* with *P*.
            3. If *exists* is **true**, then
                a Let *subElement* be the result of calling the [[Get]] internal method of *E* with argument *P*.

b Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*), Property Descriptor {[[Value]]: *subElement*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.

4. Increase *n* by 1.
5. Increase *k* by 1.

c. Else, *E* is not an Array

i. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*), Property Descriptor {[[Value]]: *E*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.

ii. Increase *n* by 1.

6. Return *A*.

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 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 a host object is implementation-dependent.

15.4.4.5 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. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument **`"length"`**.
3. Let *len* be ToUint32(*lenVal*).
4. If *separator* is **undefined**, let *separator* be the single-character String **`","`**.
5. Let *sep* be ToString(*separator*).
6. If *len* is zero, return the empty String.
7. Let *element0* be the result of calling the [[Get]] internal method of *O* with argument **`"0"`**.
8. If *element0* is **undefined** or **null**, let *R* be the empty String; otherwise, Let *R* be ToString(*element0*).
9. Let *k* be **`1`**.
10. Repeat, while *k* < *len*
    a. Let *S* be the String value produced by concatenating *R* and *sep*.
    b. Let *element* be the result of calling the [[Get]] internal method of *O* with argument ToString(*k*).
    c. If *element* is **undefined** or **null**, Let *next* be the empty String; otherwise, let *next* be ToString(*element*).
    d. Let *R* be a String value produced by concatenating *S* and *next*.
    e. Increase *k* by 1.
11. 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 a host object is implementation-dependent.

15.4.4.6 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. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument "**`length`**".
3. Let *len* be ToUint32(*lenVal*).
4. If *len* is zero,
    a. Call the [[Put]] internal method of *O* with arguments "**`length`**", 0, and **true**.

b. Return **undefined**.

5. Else, *len* > 0
   a. Let *indx* be ToString(*len*–1).
   b. Let *element* be the result of calling the [[Get]] internal method of *O* with argument *indx*.
   c. Call the [[Delete]] internal method of *O* with arguments *indx* and **true**.
   d. Call the [[Put]] internal method of *O* with arguments "**length**", *indx*, and **true**.
   e. 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 a host object is implementation-dependent.

15.4.4.7 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. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument "**length**".
3. Let *n* be ToUint32(*lenVal*).
4. Let *items* be an internal List whose elements are, in left to right order, the arguments that were passed to this function invocation.
5. Repeat, while *items* is not empty
   a. Remove the first element from *items* and let *E* be the value of the element.
   b. Call the [[Put]] internal method of *O* with arguments ToString(*n*), *E*, and **true**.
   c. Increase *n* by 1.
6. Call the [[Put]] internal method of *O* with arguments "**length**", *n*, and **true**.
7. 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 a host object is implementation-dependent.

15.4.4.8 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. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument "**length**".
3. Let *len* be ToUint32(*lenVal*).
4. Let *middle* be floor(*len*/2).
5. Let *lower* be **0**.
6. Repeat, while *lower* ≠ *middle*
   a. Let *upper* be *len*– *lower* –1.
   b. Let *upperP* be ToString(*upper*).
   c. Let *lowerP* be ToString(*lower*).
   d. Let *lowerValue* be the result of calling the [[Get]] internal method of *O* with argument *lowerP*.
   e. Let *upperValue* be the result of calling the [[Get]] internal method of *O* with argument *upperP* .
   f. Let *lowerExists* be the result of calling the [[HasProperty]] internal method of *O* with argument *lowerP*.
   g. Let *upperExists* be the result of calling the [[HasProperty]] internal method of *O* with argument *upperP*.
   h. If *lowerExists* is **true** and *upperExists* is **true**, then

i. Call the [[Put]] internal method of *O* with arguments *lowerP*, *upperValue*, and **true** .
ii. Call the [[Put]] internal method of *O* with arguments *upperP*, *lowerValue*, and **true** .

i. Else if *lowerExists* is **false** and *upperExists* is **true**, then
i. Call the [[Put]] internal method of *O* with arguments *lowerP*, *upperValue*, and **true** .
ii. Call the [[Delete]] internal method of *O*, with arguments *upperP* and **true**.

j. Else if *lowerExists* is **true** and *upperExists* is **false**, then
i. Call the [[Delete]] internal method of *O*, with arguments *lowerP* and **true** .
ii. Call the [[Put]] internal method of *O* with arguments *upperP*, *lowerValue*, and **true** .

k. Else, both *lowerExists* and *upperExists* are **false**
i. No action is required.

l. Increase *lower* by 1.

7. 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 a host object is implementation-dependent.

15.4.4.9 Array.prototype.shift ( )

The first 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. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument "**length**".
3. Let *len* be ToUint32(*lenVal*).
4. If *len* is zero, then
   a. Call the [[Put]] internal method of *O* with arguments "**length**", 0, and **true**.
   b. Return **undefined**.
5. Let *first* be the result of calling the [[Get]] internal method of *O* with argument "**0**".
6. Let *k* be 1.
7. Repeat, while *k* < *len*
   a. Let *from* be ToString(*k*).
   b. Let *to* be ToString(*k*–1).
   c. Let *fromPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *from*.
   d. If *fromPresent* is **true**, then
      i. Let *fromVal* be the result of calling the [[Get]] internal method of *O* with argument *from*.
      ii. Call the [[Put]] internal method of *O* with arguments *to*, *fromVal*, and **true**.
   e. Else, *fromPresent* is **false**
      i. Call the [[Delete]] internal method of *O* with arguments *to* and **true**.
   f. Increase *k* by 1.
8. Call the [[Delete]] internal method of *O* with arguments ToString(*len*–1) and **true**.
9. Call the [[Put]] internal method of *O* with arguments "**length**", (*len*–1) , and **true**.
10. 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 a host object is implementation-dependent.

15.4.4.10 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. Let *A* be a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
3. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument "**length**".

4. Let *len* be ToUint32(*lenVal*).
5. Let *relativeStart* be ToInteger(*start*).
6. If *relativeStart* is negative, let *k* be max((*len* + *relativeStart*),0); else let *k* be min(*relativeStart*, *len*).
7. If *end* is **undefined**, let *relativeEnd* be *len*; else let *relativeEnd* be ToInteger(*end*).
8. If *relativeEnd* is negative, let *final* be max((*len* + *relativeEnd*),0); else let *final* be min(*relativeEnd*, *len*).
9. Let *n* be 0.
10. Repeat, while *k* < *final*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
    c. If *kPresent* is **true**, then
        i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        ii. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*), Property Descriptor {[[Value]]: *kValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    d. Increase *k* by 1.
    e. Increase *n* by 1.
11. Return *A*.

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 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 a host object is implementation-dependent.

15.4.4.11 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 calling the [[Get]] internal method of *obj* with argument "**`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 value of the [[Prototype]] internal property 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 calling the [[HasProperty]] internal method of *proto* with argument 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 [[Extensible]] internal property of *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]] , [[Put]], and [[Delete]] internal methods of *obj* and to SortCompare (described below), where the first argument for each call to [[Get]], [[Put]], or [[Delete]] is a nonnegative integer less than *len* and where the arguments for calls to SortCompare are results of previous calls to the [[Get]] internal method. The throw argument to the [[Put]] and [[Delete]] internal methods will be the value **true**. If *obj* is not sparse then [[Delete]] must not be called.
2. Return *obj*.

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.

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.

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 calling the [[HasProperty]] internal method of *obj* with argument *jString*.
4. Let *hask* be the result of calling the [[HasProperty]] internal method of *obj* with argument *kString*.
5. If *hasj* and *hask* are both **false**, then return **+0**.
6. If *hasj* is **false**, then return 1.
7. If *hask* is **false**, then return –1.
8. Let *x* be the result of calling the [[Get]] internal method of *obj* with argument *jString*.
9. Let *y* be the result of calling the [[Get]] internal method of *obj* with argument *kString*.
10. If *x* and *y* are both **undefined**, return **+0**.
11. If *x* is **undefined**, return 1.
12. If *y* is **undefined**, return –1.
13. If the argument *comparefn* is not **undefined**, then
    a. If IsCallable(*comparefn*) is **false**, throw a **TypeError** exception.
    b. Return the result of calling the [[Call]] internal method of *comparefn* passing **undefined** as the **this** value and with arguments *x* and *y*.
14. Let *xString* be ToString(*x*).
15. Let *yString* be ToString(*y*).
16. If *xString* < *yString*, return –1.
17. If *xString* > *yString*, return 1.
18. 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 a host object is implementation-dependent.

15.4.4.12 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. Let *A* be a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
3. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument **"length"**.
4. Let *len* be ToUint32(*lenVal*).
5. Let *relativeStart* be ToInteger(*start*).
6. If *relativeStart* is negative, let *actualStart* be max((*len* + *relativeStart*),0); else let *actualStart* be min(*relativeStart*, *len*).
7. Let *actualDeleteCount* be min(max(ToInteger(*deleteCount*),0), *len* – *actualStart*).
8. Let *k* be 0.
9. Repeat, while *k* < *actualDeleteCount*
    a. Let *from* be ToString(*actualStart*+*k*).
    b. Let *fromPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *from*.
    c. If *fromPresent* is **true**, then
        i. Let *fromValue* be the result of calling the [[Get]] internal method of *O* with argument *from*.
        ii. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*k*), Property Descriptor {[[Value]]: *fromValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    d. Increment *k* by 1.
10. 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.
11. Let *itemCount* be the number of elements in *items*.
12. 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 calling the [[HasProperty]] internal method of *O* with argument *from*.
        iv. If *fromPresent* is **true**, then
            1. Let *fromValue* be the result of calling the [[Get]] internal method of *O* with argument *from*.
            2. Call the [[Put]] internal method of *O* with arguments *to*, *fromValue*, and **true**.
        v. Else, *fromPresent* is **false**
            1. Call the [[Delete]] internal method of *O* with arguments *to* and **true**.
        vi. Increase k by 1.
    c. Let *k* be *len*.
    d. Repeat, while *k* > (*len* – *actualDeleteCount* + *itemCount*)
        i. Call the [[Delete]] internal method of *O* with arguments ToString(*k*–1) and **true**.
        ii. Decrease *k* by 1.
13. 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 calling the [[HasProperty]] internal method of *O* with argument *from*.
iv. If *fromPresent* is **true**, then
1. Let *fromValue* be the result of calling the [[Get]] internal method of *O* with argument *from*.
2. Call the [[Put]] internal method of *O* with arguments *to*, *fromValue*, and **true**.
v. Else, *fromPresent* is **false**
1. Call the [[Delete]] internal method of *O* with argument *to* and **true**.
vi. Decrease *k* by 1.

14. Let *k* be *actualStart*.
15. Repeat, while *items* is not empty
a. Remove the first element from *items* and let *E* be the value of that element.
b. Call the [[Put]] internal method of *O* with arguments ToString(*k*), *E*, and **true**.
c. Increase *k* by 1.
16. Call the [[Put]] internal method of *O* with arguments "**length**", (*len* – *actualDeleteCount* + *itemCount*), and **true**.
17. Return *A*.

The **length** property of the **splice** method is **2**.

NOTE 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 a host object is implementation-dependent.

15.4.4.13 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. Let *lenVal* be the result of calling the [[Get]] internal method of *O* with argument "**length**".
3. Let *len* be ToUint32(*lenVal*).
4. Let *argCount* be the number of actual arguments.
5. Let *k* be *len*.
6. 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 calling the [[HasProperty]] internal method of *O* with argument *from*.
d. If *fromPresent* is **true**, then
i. Let *fromValue* be the result of calling the [[Get]] internal method of *O* with argument *from*.
ii. Call the [[Put]] internal method of *O* with arguments *to*, *fromValue*, and **true**.
e. Else, *fromPresent* is **false**
i. Call the [[Delete]] internal method of *O* with arguments *to*, and **true**.
f. Decrease *k* by 1.
7. Let *j* be 0.
8. Let *items* be an internal List whose elements are, in left to right order, the arguments that were passed to this function invocation.
9. Repeat, while *items* is not empty
a. Remove the first element from *items* and let *E* be the value of that element.
b. Call the [[Put]] internal method of *O* with arguments ToString(*j*), *E*, and **true**.
c. Increase *j* by 1.
10. Call the [[Put]] internal method of *O* with arguments "**length**", *len*+*argCount*, and **true**.
11. 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 a host object is implementation-dependent.

15.4.4.14 Array.prototype.indexOf (searchElement [, fromIndex])

indexOf compares *searchElement* to the elements of the array, in ascending order, using the internal Strict Equality Comparison Algorithm (11.9.6), 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. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **"length"**.
3. Let *len* be ToUint32(*lenValue*).
4. If *len* is 0, return -1.
5. If argument *fromIndex* was passed let *n* be ToInteger(*fromIndex*); else let *n* be 0.
6. If $n \geq len$, return -1.
7. If $n \geq 0$, then
    a. Let *k* be *n*.
8. Else, $n<0$
    a. Let *k* be *len* - abs(*n*).
    b. If *k* is less than 0, then let *k* be 0.
9. Repeat, while $k<len$
    a. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument ToString(*k*).
    b. If *kPresent* is **true**, then
        i. Let *elementK* be the result of calling the [[Get]] internal method of *O* with the argument ToString(*k*).
        ii. Let *same* be the result of applying the Strict Equality Comparison Algorithm to *searchElement* and *elementK*.
        iii. If *same* is **true**, return *k*.
    c. Increase *k* by 1.
10. 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 a host object is implementation-dependent.

15.4.4.15 Array.prototype.lastIndexOf (searchElement [, fromIndex])

lastIndexOf compares *searchElement* to the elements of the array in descending order using the internal Strict Equality Comparison Algorithm (11.9.6), 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. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **`"length"`**.
3. Let *len* be ToUint32(*lenValue*).
4. If *len* is 0, return -1.
5. If argument *fromIndex* was passed let *n* be ToInteger(*fromIndex*); else let *n* be *len*-1.
6. If $n \geq 0$, then let *k* be min(*n*, *len* – 1).
7. Else, $n < 0$
    a. Let *k* be *len* - abs(*n*).
8. Repeat, while $k \geq 0$
    a. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument ToString(*k*).
    b. If *kPresent* is **true**, then
        i. Let *elementK* be the result of calling the [[Get]] internal method of *O* with the argument ToString(*k*).
        ii. Let *same* be the result of applying the Strict Equality Comparison Algorithm to *searchElement* and *elementK*.
        iii. If *same* is **true,** return *k*.
    c. Decrease *k* by 1.
9. Return -1.

The **`length`** property of the **`lastIndexOf`** method is **1**.

NOTE The **`lastIndexOf`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **`lastIndexOf`** function can be applied successfully to a host object is implementation-dependent.

15.4.4.16 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. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **`"length"`**.
3. Let *len* be ToUint32(*lenValue*).
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
6. Let *k* be 0.

7. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
    c. If *kPresent* is **true**, then
        i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        ii. Let *testResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as the **this** value and argument list containing *kValue*, *k*, and *O*.
        iii. If ToBoolean(*testResult)* is **false**, return **false**.
    d. Increase *k* by 1.
8. 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 a host object is implementation-dependent.

15.4.4.17 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 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. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **`"length"`**.
3. Let *len* be ToUint32(*lenValue*).
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
6. Let *k* be 0.
7. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
    c. If *kPresent* is **true**, then
        i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        ii. Let *testResult* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as the **this** value and argument list containing *kValue*, *k*, and *O*.
        iii. If ToBoolean(*testResult)* is **true**, return **true**.
    d. Increase *k* by 1.
8. 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 a host object is implementation-dependent.

15.4.4.18 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.

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. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **"length"**.
3. Let *len* be ToUint32(*lenValue*).
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
6. Let *k* be 0.
7. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
    c. If *kPresent* is **true**, then
        i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        ii. Call the [[Call]] internal method of *callbackfn* with *T* as the **this** value and argument list containing *kValue*, *k*, and *O*.
    d. Increase *k* by 1.
8. 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 a host object is implementation-dependent.

15.4.4.19 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:

1. Let *O* be the result of calling ToObject passing the **this** value as the argument.
2. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **"length"**.
3. Let *len* be ToUint32(*lenValue*).
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
6. Let *A* be a new array created as if by the expression **new Array(***len***)** where **Array** is the standard built-in constructor with that name and *len* is the value of *len*.
7. Let *k* be 0.
8. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
    c. If *kPresent* is **true**, then
        i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        ii. Let *mappedValue* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as the **this** value and argument list containing *kValue*, *k*, and *O*.
        iii. Call the [[DefineOwnProperty]] internal method of *A* with arguments *Pk*, Property Descriptor {[[Value]]: *mappedValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    d. Increase *k* by 1.
9. 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 a host object is implementation-dependent.

15.4.4.20 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. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **"length"**.
3. Let *len* be ToUint32(*lenValue*).
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If *thisArg* was supplied, let *T* be *thisArg*; else let *T* be **undefined**.
6. Let *A* be a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
7. Let *k* be 0.
8. Let *to* be 0.
9. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
    c. If *kPresent* is **true**, then
        i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        ii. Let *selected* be the result of calling the [[Call]] internal method of *callbackfn* with *T* as the **this** value and argument list containing *kValue*, *k*, and *O*.
        iii. If ToBoolean(*selected*) is **true**, then
            1. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*to*), Property Descriptor {[[Value]]: *kValue*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
            2. Increase *to* by 1.
    d. Increase *k* by 1.
10. 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 a host object is implementation-dependent.

15.4.4.21 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. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **"length"**.
3. Let *len* be ToUint32(*lenValue*).
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If len is 0 and *initialValue* is not present, throw a **TypeError** exception.
6. Let *k* be 0.

7. If *initialValue* is present, then
    a. Set *accumulator* to *initialValue*.
8. 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 calling the [[HasProperty]] internal method of *O* with argument *Pk*.
        iii. If *kPresent* is **true**, then
            1. Let *accumulator* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        iv. Increase *k* by 1.
    c. If *kPresent* is **false**, throw a **TypeError** exception.
9. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
    c. If *kPresent* is **true**, then
        i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        ii. Let *accumulator* be the result of calling the [[Call]] internal method of *callbackfn* with **undefined** as the **this** value and argument list containing *accumulator*, *kValue*, *k*, and *O*.
    d. Increase *k* by 1.
10. 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 a host object is implementation-dependent.

15.4.4.22 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. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **`"length"`**.
3. Let *len* be ToUint32(*lenValue*).
4. If IsCallable(*callbackfn*) is **false**, throw a **TypeError** exception.
5. If len is 0 and *initialValue* is not present, throw a **TypeError** exception.
6. Let *k* be *len*-1.
7. If *initialValue* is present, then

a. Set *accumulator* to *initialValue*.
8. 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 calling the [[HasProperty]] internal method of *O* with argument *Pk*.
      iii. If *kPresent* is **true**, then
         1. Let *accumulator* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
      iv. Decrease *k* by 1.
   c. If *kPresent* is **false**, throw a **TypeError** exception.
9. Repeat, while $k \geq 0$
   a. Let *Pk* be ToString(*k*).
   b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
   c. If *kPresent* is **true**, then
      i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
      ii. Let *accumulator* be the result of calling the [[Call]] internal method of *callbackfn* with **undefined** as the **this** value and argument list containing *accumulator*, *kValue*, *k*, and *O*.
   d. Decrease *k* by 1.
10. 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 a host object is implementation-dependent.

15.4.5 Properties of Array Instances

Array instances inherit properties from the Array prototype object and their [[Class]] internal property value is **`"Array"`**. Array instances also have the following properties.

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:

1. 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.
2. Let *oldLen* be *oldLenDesc*.[[Value]].
3. If *P* is **`"length"`**, then
   a. If the [[Value]] field of *Desc* is absent, then
      i. Return the result of calling the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing **`"length"`**, *Desc*, and *Throw* as arguments.
   b. Let *newLenDesc* be a copy of *Desc*.
   c. Let *newLen* be ToUint32(*Desc*.[[Value]]).
   d. If *newLen* is not equal to ToNumber(*Desc*.[[Value]]), throw a **RangeError** exception.
   e. Set *newLenDesc*.[[Value] to *newLen*.
   f. If *newLen* ≥*oldLen*, then

i. Return the result of calling the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing "**length**", *newLenDesc*, and *Throw* as arguments.

g. Reject if *oldLenDesc*.[[Writable]] is **false**.
h. If *newLenDesc*.[[Writable]] is absent or has the value **true**, let *newWritable* be **true**.
i. Else,
   i. Need to defer setting the [[Writable]] attribute to **false** in case any elements cannot be deleted.
   ii. Let *newWritable* be **false**.
   iii. Set *newLenDesc*.[[Writable] to **true**.
j. Let *succeeded* be the result of calling the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing "**length**", *newLenDesc*, and *Throw* as arguments.
k. If *succeeded* is **false**, return **false**.
l. While *newLen* < *oldLen* repeat,
   i. Set *oldLen* to *oldLen* – 1.
   ii. Let *deleteSucceeded* be the result of calling the [[Delete]] internal method of *A* passing ToString(*oldLen*) and **false** as arguments.
   iii. If *deleteSucceeded* is **false**, then
      1. Set *newLenDesc*.[[Value] to *oldLen+1*.
      2. If *newWritable* is **false**, set *newLenDesc*.[[Writable] to **false**.
      3. Call the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing "**length**", *newLenDesc*, and **false** as arguments.
      4. Reject.
m. If *newWritable* is **false**, then
   i. Call the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing "**length**", Property Descriptor{[[Writable]]: **false**}, and **false** as arguments. This call will always return **true**.
n. Return **true**.

4. Else if *P* is an array index (15.4), then
   a. Let *index* be ToUint32(*P*).
   b. Reject if *index* ≥ *oldLen* and *oldLenDesc*.[[Writable]] is **false**.
   c. Let *succeeded* be the result of calling the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing *P*, *Desc*, and **false** as arguments.
   d. Reject if *succeeded* is **false**.
   e. If *index* ≥ *oldLen*
      i. Set *oldLenDesc*.[[Value]] to *index* + 1.
      ii. Call the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing "**length**", *oldLenDesc*, and **false** as arguments. This call will always return **true**.
   f. Return **true**.
5. Return the result of calling the default [[DefineOwnProperty]] internal method (8.12.9) on *A* passing *P*, *Desc*, and *Throw* as arguments.

15.4.5.2 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 15.4.5.1.

15.5 String Objects

15.5.1 The String Constructor Called as a Function

When **String** is called as a function rather than as a constructor, it performs a type conversion.

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.

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 object.

15.5.2.1 new String ([value])

The [[Prototype]] internal property of the newly constructed object is set to the standard built-in String prototype object that is the initial value of **String.prototype** (15.5.3.1).

The [[Class]] internal property of the newly constructed object is set to **"String"**.

The [[Extensible]] internal property of the newly constructed object is set to **true**.

The [[PrimitiveValue]] internal property of the newly constructed object is set to ToString(*value*), or to the empty String if *value* is not supplied.

15.5.3 Properties of the String Constructor

The value of the [[Prototype]] internal property of the String constructor is the standard built-in Function prototype object (15.3.4).

Besides the internal properties and the **length** property (whose value is **1**), the String constructor has the following properties:

15.5.3.1 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** }.

15.5.3.2 String.fromCharCode ([char0 [, char1 [, …]]])

Returns a String value containing as many characters as the number of arguments. 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 integer as the code unit value of a character. If no arguments are supplied, the result is the empty String.

The **length** property of the **fromCharCode** function is **1**.

15.5.4 Properties of the String Prototype Object

The String prototype object is itself a String object (its [[Class]] is **"String"**) whose value is an empty String.

The value of the [[Prototype]] internal property of the String prototype object is the standard built-in Object prototype object (15.2.4).

15.5.4.1 String.prototype.constructor

The initial value of **String.prototype.constructor** is the built-in **String** constructor.

15.5.4.2 String.prototype.toString ( )

Returns this String value. (Note that, for a String object, the **`toString`** method happens to return the same thing as the **`valueOf`** method.)

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 method.

15.5.4.3 String.prototype.valueOf ( )

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 other kinds of objects for use as a method.

15.5.4.4 String.prototype.charAt (pos)

Returns a String containing the character at position *pos* in the String resulting from converting this object to a String. If there is no character 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. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *position* be ToInteger(*pos*).
4. Let *size* be the number of characters in *S*.
5. If *position* < 0 or *position* ≥ *size*, return the empty String.
6. Return a String of length 1, containing one character from *S*, namely the character at position *position*, where the first (leftmost) character 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.

15.5.4.5 String.prototype.charCodeAt (pos)

Returns a Number (a nonnegative integer less than 2^{16}) representing the code unit value of the character at position *pos* in the String resulting from converting this object to a String. If there is no character at that position, the result is **NaN**.

When the **`charCodeAt`** method is called with one argument *pos*, the following steps are taken:

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *position* be ToInteger(*pos*).
4. Let *size* be the number of characters in *S*.
5. If *position* < 0 or *position* ≥ *size*, return **NaN**.
6. Return a value of Number type, whose value is the code unit value of the character at position *position* in the String *S*, where the first (leftmost) character in *S* is considered to be at position 0, the next one at position 1, and so on.

NOTE The **`charCodeAt`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.4.6 String.prototype.concat ([string1 [, string2 [, …]]])

When the **concat** method is called with zero or more arguments *string1*, *string2*, etc., it returns a String consisting of the characters of this object (converted to a String) followed by the characters of each of *string1*, *string2*, etc. (where each argument is converted to a String). The result is a String value, not a String object. The following steps are taken:

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *args* be an internal list that is a copy of the argument list passed to this function.
4. Let *R* be *S*.
5. Repeat, while *args* is not empty
    a. Remove the first element from *args* and let *next* be the value of that element.
    b. Let *R* be the String value consisting of the characters in the previous value of *R* followed by the characters of ToString(*next*).
6. Return *R*.

The **length** property of the **concat** method is **1**.

NOTE The **concat** function is intentionally generic; it does not require that its **this** value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.4.7 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. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *searchStr* be ToString(*searchString*).
4. Let *pos* be ToInteger(*position*). (If *position* is **undefined**, this step produces the value **0**).
5. Let *len* be the number of characters in *S*.
6. Let *start* be min(max(*pos*, 0), *len*).
7. Let *searchLen* be the number of characters in *searchStr*.
8. Return the smallest possible integer *k* not smaller than *start* such that *k*+ *searchLen* is not greater than *len*, and for all nonnegative integers *j* less than *searchLen*, the character at position *k*+*j* of *S* is the same as the character at position *j* of *searchStr*; but if there is no such integer *k*, then return the value **-1**.

The **length** property of the **indexOf** method is **1**.

NOTE The **indexOf** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.8 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. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *searchStr* be ToString(*searchString*).

4. Let *numPos* be ToNumber(*position*). (If *position* is **undefined**, this step produces the value **NaN**).
5. If *numPos* is **NaN**, let *pos* be +∞; otherwise, let *pos* be ToInteger(*numPos*).
6. Let *len* be the number of characters in *S*.
7. Let *start* min(max(*pos*, 0), *len*).
8. Let *searchLen* be the number of characters in *searchStr*.
9. Return the largest possible nonnegative integer *k* not larger than *start* such that *k*+ *searchLen* is not greater than *len*, and for all nonnegative integers *j* less than *searchLen*, the character at position *k*+*j* of *S* is the same as the character at position *j* of *searchStr*; but if there is no such integer *k*, then return the value **-1**.

The **length** property of the **lastIndexOf** method is **1**.

NOTE The **lastIndexOf** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.9 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. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *That* be ToString(*that*).

The **localeCompare** method, if considered as a function of two arguments **this** and *that*, is a consistent comparison function (as defined in 15.4.4.11) 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.

15.5.4.10 String.prototype.match (regexp)

When the **match** method is called with argument *regexp*, the following steps are taken:

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. If Type(*regexp*) is Object and the value of the [[Class]] internal property of *regexp* is **"RegExp"**, then let *rx* be *regexp*;
4. Else, let *rx* be a new RegExp object created as if by the expression **new RegExp(***regexp***)** where **RegExp** is the standard built-in constructor with that name.
5. Let *global* be the result of calling the [[Get]] internal method of *rx* with argument **"global"**.
6. Let *exec* be the standard built-in function **RegExp.prototype.exec** (see 15.10.6.2)
7. If *global* is not **true**, then
    a. Return the result of calling the [[Call]] internal method of *exec* with *rx* as the **this** value and argument list containing *S*.
8. Else, *global* is **true**
    a. Call the [[Put]] internal method of *rx* with arguments **"lastIndex"** and 0.
    b. Let *A* be a new array created as if by the expression **new Array()** where **Array** is the standard built-in constructor with that name.
    c. Let *previousLastIndex* be 0.
    d. Let *n* be 0.
    e. Let *lastMatch* be **true**.
    f. Repeat, while *lastMatch* is **true**
        i. Let *result* be the result of calling the [[Call]] internal method of *exec* with *rx* as the **this** value and argument list containing *S*.
        ii. If *result* is **null**, then set *lastMatch* to **false**.
        iii. Else, *result* is not **null**
            1. Let *thisIndex* be the result of calling the [[Get]] internal method of *rx* with argument **"lastIndex"**.
            2. If *thisIndex* = *previousLastIndex* then
                a Call the [[Put]] internal method of *rx* with arguments **"lastIndex"** and *thisIndex*+1.
                b Set *previousLastIndex* to *thisIndex*+1.
            3. Else, set *previousLastIndex* to *thisIndex*.
            4. Let *matchStr* be the result of calling the [[Get]] internal method of *result* with argument **"0"**.
            5. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*n*), the Property Descriptor {[[Value]]: *matchStr*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[configurable]]: **true**}, and **false**.
            6. Increment *n*.
    g. If *n* = 0, then return **null**.
    h. Return *A*.

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.

15.5.4.11 String.prototype.replace (searchValue, replaceValue)

First set *string* according to the following steps:

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *string* be the result of calling ToString, giving it the **this** value as its argument.

If *searchValue* is a regular expression (an object whose [[Class]] internal property is **"RegExp"**), do the following: 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 **String.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).

If *searchValue* is not a regular expression, let *searchString* be ToString(*searchValue*) and search *string* for the first occurrence of *searchString*. Let *m* be 0.

If *replaceValue* is a function, then 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.

Otherwise, 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 characters in *newstring* by replacement text as specified in Table 22. 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.

Table 22 — Replacement Text Symbol Substitutions

Characters	Replacement text
$$	**$**
$&	The matched substring.
$`	The portion of *string* that precedes the matched substring.
$'	The portion of *string* that follows the matched substring.
$n	The n^{th} capture, where *n* is a single digit in the range 1 to 9 and **$***n* is not followed by a decimal digit. If *n*≤*m* and the *n*th capture is **undefined**, use the empty String instead. If *n*>*m*, the result is implementation-defined.
$nn	The nn^{th} capture, where *nn* is a two-digit decimal number in the range 01 to 99. If *nn*≤*m* and the nn^{th} capture is **undefined**, use the empty String instead. If *nn*>*m*, the result is implementation-defined.

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.

15.5.4.12 String.prototype.search (regexp)

When the search method is called with argument *regexp*, the following steps are taken:

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *string* be the result of calling ToString, giving it the **this** value as its argument.
3. If Type(*regexp*) is Object and the value of the [[Class]] internal property of *regexp* is **`"RegExp"`**, then let *rx* be *regexp*;
4. Else, let *rx* be a new RegExp object created as if by the expression **`new RegExp(`***regexp***`)`** where **`RegExp`** is the standard built-in constructor with that name.
5. 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. The **`lastIndex`** and **`global`** properties of *regexp* are ignored when performing the search. The **`lastIndex`** property of *regexp* is left unchanged.
6. Return *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.

15.5.4.13 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 character position *start* and running to, but not including, character position *end* (or through the end of the String if *end* is **undefined**). If *start* is negative, it is treated as *sourceLength*+*start* where *sourceLength* is the length of the String. If *end* is negative, it is treated as *sourceLength*+*end* where

sourceLength is the length of the String. The result is a String value, not a String object. The following steps are taken:

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *len* be the number of characters in *S*.
4. Let *intStart* be ToInteger(*start*).
5. If *end* is **undefined**, let *intEnd* be *len*; else let *intEnd* be ToInteger(*end*).
6. If *intStart* is negative, let *from* be max(*len* + *intStart*,0); else let *from* be min(*intStart*, *len*).
7. If *intEnd* is negative, let *to* be max(*len* + *intEnd*,0); else let *to* be min(*intEnd*, *len*).
8. Let *span* be max(*to* – *from*,0).
9. Return a String containing *span* consecutive characters from *S* beginning with the character at position *from*.

The **`length`** property of the **`slice`** method is **2**.

NOTE The **`slice`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore it can be transferred to other kinds of objects for use as a method.

15.5.4.14 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 (i.e., an object 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 characters; the length of the result array equals the length of the String, and each substring contains one character.) If *separator* is a regular expression, only the first match at a given position of the **this** String is considered, even if backtracking could yield a non-empty-substring match at that position. (For example, **`"ab".split(/a*?/)`** evaluates to the array **`["a","b"]`**, while **`"ab".split(/a*/)`** evaluates to the array **`["","b"]`**.)

If the **this** object is (or converts to) the empty String, the result depends on whether *separator* can match the empty String. If it can, the result array contains no elements. Otherwise, the result array contains one element, which is the empty String.

If *separator* is a regular expression that contains capturing parentheses, then each time *separator* is matched the results (including any **undefined** results) of the capturing parentheses are spliced into the output array. For example,

```
"A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?([^<>]+)>/)
```

evaluates to the array

```
["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. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *A* be a new array created as if by the expression **`new Array()`** where **`Array`** is the standard built-in constructor with that name.
4. Let *lengthA* be 0.

5. If *limit* is **undefined**, let *lim* = 2^{32}–1; else let *lim* = ToUint32(*limit*).
6. Let *s* be the number of characters in *S*.
7. Let *p* = 0.
8. If *separator* is a RegExp object (its [[Class]] is **"RegExp"**), let *R* = *separator*; otherwise let *R* = ToString(*separator*).
9. If *lim* = 0, return *A*.
10. If *separator* is **undefined**, then
    a. Call the [[DefineOwnProperty]] internal method of *A* with arguments **"0"**, Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    b. Return *A*.
11. If *s* = 0, then
    a. Call *SplitMatch*(*S*, 0, *R*) and let *z* be its MatchResult result.
    b. If *z* is not **failure**, return *A*.
    c. Call the [[DefineOwnProperty]] internal method of *A* with arguments **"0"**, Property Descriptor {[[Value]]: *S*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
    d. Return *A*.
12. Let *q* = *p*.
13. Repeat, while *q* ≠ *s*
    a. Call *SplitMatch*(*S*, *q*, *R*) and let *z* be its MatchResult result.
    b. If *z* is **failure**, then let *q* = *q*+1.
    c. 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 characters at positions *p* (inclusive) through *q* (exclusive).
            2. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*), Property Descriptor {[[Value]]: *T*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
            3. Increment *lengthA* by 1.
            4. If *lengthA* = *lim*, return *A*.
            5. Let p = e.
            6. Let *i* = 0.
            7. Repeat, while *i* is not equal to the number of elements in *cap*.
                a Let *i* = *i*+1.
                b Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*), Property Descriptor {[[Value]]: *cap*[*i*], [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
                c Increment *lengthA* by 1.
                d If *lengthA* = *lim*, return *A*.
            8. Let *q* = *p*.
14. Let *T* be a String value equal to the substring of *S* consisting of the characters at positions *p* (inclusive) through *s* (exclusive).
15. Call the [[DefineOwnProperty]] internal method of *A* with arguments ToString(*lengthA*), Property Descriptor {[[Value]]: *T*, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
16. Return *A*.

The abstract operation *SplitMatch* takes three parameters, a String *S*, an integer *q*, and a String or RegExp *R*, and performs the following in order to return a MatchResult (see 15.10.2.1):

1. If *R* is a RegExp object (its [[Class]] is **"RegExp"**), then
    a. Call the [[Match]] internal method of *R* giving it the arguments *S* and *q*, and return the MatchResult result.
2. Type(*R*) must be String. Let *r* be the number of characters in *R*.
3. Let *s* be the number of characters in *S*.
4. If *q*+*r* > *s* then return the MatchResult **failure**.
5. If there exists an integer *i* between 0 (inclusive) and *r* (exclusive) such that the character at position *q*+*i* of *S* is different from the character at position *i* of *R*, then return **failure**.
6. Let *cap* be an empty array of captures (see 15.10.2.1).
7. Return the State (*q*+*r*, *cap*). (see 15.10.2.1)

The **length** property of the **split** method is **2**.

NOTE 1 The **split** method ignores the value of separator.**global** for separators that are RegExp objects.

NOTE 2 The **split** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.15 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 character position *start* and running to, but not including, character 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. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *len* be the number of characters in *S*.
4. Let *intStart* be ToInteger(*start*).
5. If *end* is **undefined**, let *intEnd* be *len*; else let *intEnd* be ToInteger(*end*).
6. Let *finalStart* be min(max(*intStart*, 0), *len*).
7. Let *finalEnd* be min(max(*intEnd*, 0), *len*).
8. Let *from* be min(*finalStart*, *finalEnd*).
9. Let *to* be max(*finalStart*, *finalEnd*).
10. Return a String whose length is *to* - *from*, containing characters from *S*, namely the characters 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.

15.5.4.16 String.prototype.toLowerCase ( )

The following steps are taken:

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *L* be a String 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.
4. Return *L*.

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.

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 in Unicode 2.1.8 and later).

NOTE 1 The case mapping of some characters may produce multiple characters. In this case the result String may not be the same length as the source String. Because both **toUpperCase** and **toLowerCase** have context-sensitive

behaviour, the functions are not symmetrical. In other words, **`s.toUpperCase().toLowerCase()`** is not necessarily equal to **`s.toLowerCase()`**.

NOTE 2 The **`toLowerCase`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.17 String.prototype.toLocaleLowerCase ( )

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.

15.5.4.18 String.prototype.toUpperCase ( )

This function behaves in exactly the same way as **`String.prototype.toLowerCase`**, except that characters are mapped to their *uppercase* equivalents as specified in the Unicode Character Database.

NOTE The **`toUpperCase`** function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.19 String.prototype.toLocaleUpperCase ( )

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.

15.5.4.20 String.prototype.trim ( )

The following steps are taken:

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString, giving it the **this** value as its argument.
3. Let *T* be a String value that is a copy of *S* with both leading and trailing white space removed. The definition of white space is the union of *WhiteSpace* and *LineTerminator*.
4. 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.

15.5.5 Properties of String Instances

String instances inherit properties from the String prototype object and their [[Class]] internal property value is **"String"**. String instances also have a [[PrimitiveValue]] internal property, a **length** property, and a set of enumerable properties with array index names.

The [[PrimitiveValue]] internal property is the String value represented by this String object. The array index named properties correspond to the individual characters of the String value. A special [[GetOwnProperty]] internal method is used to specify the number, values, and attributes of the array index named properties.

15.5.5.1 length

The number of characters in the String value represented by this String object.

Once a String object is created, this property is unchanging. It has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.5.5.2 [[GetOwnProperty]] (P)

String objects use a variation of the [[GetOwnProperty]] internal method used for other native ECMAScript objects (8.12.1). This special internal method provides access to named properties corresponding to the individual characters 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:

1. Let *desc* be the result of calling the default [[GetOwnProperty]] internal method (8.12.1) on *S* with argument *P*.
2. If *desc* is not **undefined** return *desc*.
3. If ToString(abs(ToInteger(*P*))) is not the same value as *P*, return **undefined**.
4. Let *str* be the String value of the [[PrimitiveValue]] internal property of *S*.
5. Let *index* be ToInteger(*P*).
6. Let *len* be the number of characters in *str*.
7. If *len* ≤ *index*, return **undefined**.
8. Let *resultStr* be a String of length 1, containing one character from *str*, specifically the character at position *index*, where the first (leftmost) character in *str* is considered to be at position 0, the next one at position 1, and so on.
9. Return a Property Descriptor { [[Value]]: *resultStr*, [[Enumerable]]: **true**, [[Writable]]: **false**, [[Configurable]]: **false** }

15.6 Boolean Objects

15.6.1 The Boolean Constructor Called as a Function

When **Boolean** is called as a function rather than as a constructor, it performs a type conversion.

15.6.1.1 Boolean (value)

Returns a Boolean value (not a Boolean object) computed by ToBoolean(*value*).

15.6.2 The Boolean Constructor

When **Boolean** is called as part of a **new** expression it is a constructor: it initialises the newly created object.

15.6.2.1 new Boolean (value)

The [[Prototype]] internal 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"`**.

The [[PrimitiveValue]] internal property of the newly constructed Boolean object is set to ToBoolean(*value*).

The [[Extensible]] internal property of the newly constructed object is set to **true**.

15.6.3 Properties of the Boolean Constructor

The value of the [[Prototype]] internal property of the Boolean constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **`length`** property (whose value is **1**), the Boolean constructor has the following property:

15.6.3.1 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** }.

15.6.4 Properties of the Boolean Prototype Object

The Boolean prototype object is itself a Boolean object (its [[Class]] is **`"Boolean"`**) whose value is **false**.

The value of the [[Prototype]] internal property of the Boolean prototype object is the standard built-in Object prototype object (15.2.4).

15.6.4.1 Boolean.prototype.constructor

The initial value of **`Boolean.prototype.constructor`** is the built-in **`Boolean`** constructor.

15.6.4.2 Boolean.prototype.toString ( )

The following steps are taken:

1. Let *B* be the **this** value.
2. If Type(*B*) is Boolean, then let *b* be *B*.
3. Else if Type(*B*) is Object and the value of the [[Class]] internal property of *B* is **`"Boolean"`**, then let *b* be the value of the [[PrimitiveValue]] internal property of *B*.
4. Else throw a **TypeError** exception.
5. If *b* is **true**, then return **`"true"`**; else return **`"false"`**.

15.6.4.3 Boolean.prototype.valueOf ( )

The following steps are taken:

1. Let *B* be the **this** value.
2. If Type(*B*) is Boolean, then let *b* be *B*.
3. Else if Type(*B*) is Object and the value of the [[Class]] internal property of *B* is **`"Boolean"`**, then let *b* be the value of the [[PrimitiveValue]] internal property of *B*.

4. Else throw a **TypeError** exception.
5. Return *b*.

15.6.5 Properties of Boolean Instances

Boolean instances inherit properties from the Boolean prototype object and their [[Class]] internal property value is **"Boolean"**. Boolean instances also have a [[PrimitiveValue]] internal property.

The [[PrimitiveValue]] internal property is the Boolean value represented by this Boolean object.

15.7 Number Objects

15.7.1 The Number Constructor Called as a Function

When **Number** is called as a function rather than as a constructor, it performs a type conversion.

15.7.1.1 Number ([value])

Returns a Number value (not a Number object) computed by ToNumber(*value*) if *value* was supplied, else returns **+0**.

15.7.2 The Number Constructor

When **Number** is called as part of a **new** expression it is a constructor: it initialises the newly created object.

15.7.2.1 new Number ([value])

The [[Prototype]] internal 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"**.

The [[PrimitiveValue]] internal property of the newly constructed object is set to ToNumber(*value*) if *value* was supplied, else to +0.

The [[Extensible]] internal property of the newly constructed object is set to **true**.

15.7.3 Properties of the Number Constructor

The value of the [[Prototype]] internal property of the Number constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **length** property (whose value is **1**), the Number constructor has the following properties:

15.7.3.1 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** }.

15.7.3.2 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** }.

15.7.3.3 Number.MIN_VALUE

The value of **`Number.MIN_VALUE`** is the smallest positive value of the Number type, which is approximately 5×10^{-324}.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.3.4 Number.NaN

The value of **`Number.NaN`** is **NaN**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.3.5 Number.NEGATIVE_INFINITY

The value of Number.NEGATIVE_INFINITY is $-\infty$.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.3.6 Number.POSITIVE_INFINITY

The value of Number.POSITIVE_INFINITY is $+\infty$.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

15.7.4 Properties of the Number Prototype Object

The Number prototype object is itself a Number object (its [[Class]] is **`"Number"`**) whose value is +0.

The value of the [[Prototype]] internal 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 for which the value of the [[Class]] internal property is **`"Number"`**.

In the following descriptions of functions that are properties of the Number prototype object, the phrase “this Number object” refers to either the object that is the **this** value for the invocation of the function or, if Type(**this** value) is Number, an object that is created as if by the expression **`new Number(this`** value**`)`** where **`Number`** is the standard built-in constructor with that name. Also, the phrase “this Number value” refers to either the Number value represented by this Number object, that is, the value of the [[PrimitiveValue]] internal property of this Number object or the **this** value if its type is Number. A **TypeError** exception is thrown if the **this** value is neither an object for which the value of the [[Class]] internal property is **`"Number"`** or a value whose type is Number.

15.7.4.1 Number.prototype.constructor

The initial value of **`Number.prototype.constructor`** is the built-in **`Number`** constructor.

15.7.4.2 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 9.8.1.

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.

15.7.4.3 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.

15.7.4.4 Number.prototype.valueOf ( )

Returns this Number value.

The **`valueOf`** 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.

15.7.4.5 Number.prototype.toFixed (fractionDigits)

Return 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. Specifically, perform the following steps:

1. Let *f* be ToInteger(*fractionDigits*). (If *fractionDigits* is **undefined**, this step produces the value **`0`**).
2. If $f < 0$ or $f > 20$, throw a **RangeError** exception.
3. Let *x* be this Number value.
4. If *x* is **NaN**, return the String **`"NaN"`**.
5. Let *s* be the empty String.
6. If $x < 0$, then
   a. Let *s* be "**`-`**".
   b. Let $x = -x$.
7. If $x \geq 10^{21}$, then
   a. Let *m* = ToString(*x*).
8. Else, $x < 10^{21}$
   a. Let *n* be an integer for which the exact mathematical value of $n \div 10^f - x$ is as close to zero as possible. If there are two such *n*, pick the larger *n*.
   b. If $n = 0$, let *m* be the String **`"0"`**. Otherwise, let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
   c. If $f \neq 0$, then
      i. Let *k* be the number of characters in *m*.
      ii. If $k \leq f$, then
         1. Let *z* be the String consisting of $f+1-k$ occurrences of the character '`0`'.
         2. Let *m* be the concatenation of Strings *z* and *m*.

3. Let $k = f + 1$.

iii. Let *a* be the first *k*–*f* characters of *m*, and let *b* be the remaining *f* characters of *m*.

iv. Let *m* be the concatenation of the three Strings *a*, **"."**, and *b*.

9. Return the concatenation of the Strings *s* and *m*.

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"**.

15.7.4.6 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 this Number value.
2. Let *f* be ToInteger(*fractionDigits*).
3. If *x* is **NaN**, return the String **"NaN"**.
4. Let *s* be the empty String.
5. If $x < 0$, then
   a. Let *s* be **"-"**.
   b. Let $x = -x$.
6. If $x = +\infty$, then
   a. Return the concatenation of the Strings *s* and **"Infinity"**.
7. If *fractionDigits* is not **undefined** and ($f < 0$ or $f > 20$), throw a **RangeError** exception.
8. If $x = 0$, then
   a. Let $f = 0$.
   b. Let *m* be the String consisting of *f*+1 occurrences of the character '0'.
   c. Let $e = 0$.
9. Else, $x \neq 0$
   a. If fractionDigits is not **undefined**, then
      i. Let *e* and *n* be integers such that $10^f \leq n < 10^{f+1}$ and for which the exact mathematical value of $n \times 10^{e-f} - x$ 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 $f \geq 0$, $10^f \leq n < 10^{f+1}$, 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).
10. If $f \neq 0$, then
    a. Let *a* be the first character of *m*, and let *b* be the remaining *f* characters of *m*.
    b. Let *m* be the concatenation of the three Strings *a*, **"."**, and *b*.
11. If $e = 0$, then
    a. Let *c* = **"+"**.
    b. Let *d* = **"0"**.

12. Else
    a. If $e > 0$, then let c = **"+"**.
    b. Else, $e \leq 0$
        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).
13. Let m be the concatenation of the four Strings m, **"e"**, c, and d.
14. Return the concatenation of the Strings s and m.

The **length** property of the **toExponential** method is **1**.

If the **toExponential** method is called with more than one argument, then the behaviour is undefined (see clause 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 $f \geq 0$, $10^f \leq n < 10^{f+1}$, 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.

15.7.4.7 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 this Number value.
2. If *precision* is **undefined**, return ToString(x).
3. Let p be ToInteger(*precision*).
4. If x is **NaN**, return the String **"NaN"**.
5. Let s be the empty String.
6. If $x < 0$, then
    a. Let s be **"-"**.
    b. Let $x = -x$.
7. If $x = +\infty$, then
    a. Return the concatenation of the Strings s and **"Infinity"**.
8. If $p < 1$ or $p > 21$, throw a **RangeError** exception.
9. If $x = 0$, then
    a. Let m be the String consisting of p occurrences of the character '0'.
    b. Let $e = 0$.
10. Else $x \neq 0$,
    a. Let e and n be integers such that $10^{p-1} \leq n < 10^p$ and for which the exact mathematical value of $n \times 10^{e-p+1} - x$ is as close to zero as possible. If there are two such sets of e and n, pick the e and n for which $n \times 10^{e-p+1}$ is larger.
    b. Let m be the String consisting of the digits of the decimal representation of n (in order, with no leading zeroes).
    c. If $e < -6$ or $e \geq p$, then
        i. Let a be the first character of m, and let b be the remaining $p-1$ characters of m.
        ii. Let m be the concatenation of the three Strings a, **"."**, and b.
        iii. If $e = 0$, then
            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.

11. If $e = p-1$, then return the concatenation of the Strings s and m.
12. If $e \geq 0$, then
    a. Let m be the concatenation of the first $e+1$ characters of m, the character '**.**', and the remaining $p-(e+1)$ characters of m.
13. Else $e < 0$,
    a. Let m be the concatenation of the String **"0."**, $-(e+1)$ occurrences of the character '**0**', and the String m.
14. 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.

15.7.5 Properties of Number Instances

Number instances inherit properties from the Number prototype object and their [[Class]] internal property value is **"Number"**. Number instances also have a [[PrimitiveValue]] internal property.

The [[PrimitiveValue]] internal property is the Number value represented by this Number object.

15.8 The Math Object

The Math object is a single object that has some named properties, some of which are functions.

The value of the [[Prototype]] internal property of the Math object is the standard built-in Object prototype object (15.2.4). The value of the [[Class]] internal property of the Math object is **"Math"**.

The Math object does not have a [[Construct]] internal property; it is not possible to use the Math object as a constructor with the **new** operator.

The Math object does not have a [[Call]] internal property; 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.

15.8.1 Value Properties of the Math Object

15.8.1.1 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 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 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 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 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 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 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 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 operator to each of its arguments (in left-to-right order if there is more than one) and then performs a computation on the resulting Number value(s).

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 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 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 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 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 +∞, the result is an implementation-dependent approximation to +π/2.
- If *x* is –∞, the result is an implementation-dependent approximation to –π/2.

15.8.2.5 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 –π to +π.

- 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 +π/2.
- If *y*>0 and *x* is –0, the result is an implementation-dependent approximation to +π/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 +π.
- If *y* is +0 and *x*<0, the result is an implementation-dependent approximation to +π.
- 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 –π.
- If *y* is –0 and *x*<0, the result is an implementation-dependent approximation to –π.
- If *y*<0 and *x* is +0, the result is an implementation-dependent approximation to –π/2.
- If *y*<0 and *x* is –0, the result is an implementation-dependent approximation to –π/2.
- If *y*>0 and *y* is finite and *x* is +∞, the result is +0.
- If *y*>0 and *y* is finite and *x* is –∞, the result if an implementation-dependent approximation to +π.
- If *y*<0 and *y* is finite and *x* is +∞, the result is –0.
- If *y*<0 and *y* is finite and *x* is –∞, the result is an implementation-dependent approximation to –π.
- If *y* is +∞ and *x* is finite, the result is an implementation-dependent approximation to +π/2.
- If *y* is –∞ and *x* is finite, the result is an implementation-dependent approximation to –π/2.
- If *y* is +∞ and *x* is +∞, the result is an implementation-dependent approximation to +π/4.
- If *y* is +∞ and *x* is –∞, the result is an implementation-dependent approximation to +3π/4.
- If *y* is –∞ and *x* is +∞, the result is an implementation-dependent approximation to –π/4.
- If *y* is –∞ and *x* is –∞, the result is an implementation-dependent approximation to –3π/4.

15.8.2.6 ceil (x)

Returns the smallest (closest to –∞) 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 +∞, the result is +∞.
- If *x* is –∞, the result is –∞.
- 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 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 +∞, the result is NaN.
- If *x* is –∞, the result is NaN.

15.8.2.8 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 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 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 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 as in 11.8.5 except that +0 is considered to be larger than –0.

The **`length`** property of the **`max`** method is **2**.

15.8.2.12 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 as in 11.8.5 except that +0 is considered to be larger than –0.

The **`length`** property of the **`min`** method is **2**.

15.8.2.13 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)==1 and y is +∞, the result is NaN.
- If abs(x)==1 and y is –∞, the result is NaN.
- 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 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 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 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 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 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.

15.9 Date Objects

15.9.1 Overview of Date Objects and Definitions of Abstract 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:

$$\mathrm{TimeWithinDay}(t) = t \bmod \mathrm{msPerDay}$$

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

$$\begin{aligned}\mathrm{DaysInYear}(y) &= 365 \text{ if } (y \bmod 4) \neq 0\\ &= 366 \text{ if } (y \bmod 4) = 0 \text{ and } (y \bmod 100) \neq 0\\ &= 365 \text{ if } (y \bmod 100) = 0 \text{ and } (y \bmod 400) \neq 0\\ &= 366 \text{ if } (y \bmod 400) = 0\end{aligned}$$

All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year y is given by:

$$\mathrm{DayFromYear}(y) = 365 \times (y-1970) + \mathrm{floor}((y-1969)/4) - \mathrm{floor}((y-1901)/100) + \mathrm{floor}((y-1601)/400)$$

The time value of the start of a year is:

$$\mathrm{TimeFromYear}(y) = \mathrm{msPerDay} \times \mathrm{DayFromYear}(y)$$

A time value determines a year by:

$$\mathrm{YearFromTime}(t) = \text{the largest integer } y \text{ (closest to positive infinity) such that } \mathrm{TimeFromYear}(y) \leq t$$

The leap-year function is 1 for a time within a leap year and otherwise is zero:

$$\begin{aligned}\mathrm{InLeapYear}(t) &= 0 \text{ if } \mathrm{DaysInYear}(\mathrm{YearFromTime}(t)) = 365\\ &= 1 \text{ if } \mathrm{DaysInYear}(\mathrm{YearFromTime}(t)) = 366\end{aligned}$$

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:

$$\begin{aligned}\mathrm{MonthFromTime}(t) &= 0 \text{ if } 0 \leq \mathrm{DayWithinYear}(t) < 31\\ &= 1 \text{ if } 31 \leq \mathrm{DayWithinYear}(t) < 59+\mathrm{InLeapYear}(t)\\ &= 2 \text{ if } 59+\mathrm{InLeapYear}(t) \leq \mathrm{DayWithinYear}(t) < 90+\mathrm{InLeapYear}(t)\\ &= 3 \text{ if } 90+\mathrm{InLeapYear}(t) \leq \mathrm{DayWithinYear}(t) < 120+\mathrm{InLeapYear}(t)\\ &= 4 \text{ if } 120+\mathrm{InLeapYear}(t) \leq \mathrm{DayWithinYear}(t) < 151+\mathrm{InLeapYear}(t)\\ &= 5 \text{ if } 151+\mathrm{InLeapYear}(t) \leq \mathrm{DayWithinYear}(t) < 181+\mathrm{InLeapYear}(t)\\ &= 6 \text{ if } 181+\mathrm{InLeapYear}(t) \leq \mathrm{DayWithinYear}(t) < 212+\mathrm{InLeapYear}(t)\end{aligned}$$

$$
\begin{array}{lll}
= 7 & \text{if} & 212+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 243+\text{InLeapYear}(t) \\
= 8 & \text{if} & 243+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 273+\text{InLeapYear}(t) \\
= 9 & \text{if} & 273+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 304+\text{InLeapYear}(t) \\
= 10 & \text{if} & 304+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 334+\text{InLeapYear}(t) \\
= 11 & \text{if} & 334+\text{InLeapYear}(t) \leq \text{DayWithinYear}(t) < 365+\text{InLeapYear}(t)
\end{array}
$$

where

$$\text{DayWithinYear}(t) = \text{Day}(t) - \text{DayFromYear}(\text{YearFromTime}(t))$$

A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10 specifies November; and 11 specifies December. Note that $\text{MonthFromTime}(0) = 0$, 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:

$$
\begin{array}{lll}
\text{DateFromTime}(t) & = \text{DayWithinYear}(t)+1 & \text{if MonthFromTime}(t)=0 \\
& = \text{DayWithinYear}(t)-30 & \text{if MonthFromTime}(t)=1 \\
& = \text{DayWithinYear}(t)-58-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=2 \\
& = \text{DayWithinYear}(t)-89-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=3 \\
& = \text{DayWithinYear}(t)-119-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=4 \\
& = \text{DayWithinYear}(t)-150-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=5 \\
& = \text{DayWithinYear}(t)-180-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=6 \\
& = \text{DayWithinYear}(t)-211-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=7 \\
& = \text{DayWithinYear}(t)-242-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=8 \\
& = \text{DayWithinYear}(t)-272-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=9 \\
& = \text{DayWithinYear}(t)-303-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=10 \\
& = \text{DayWithinYear}(t)-333-\text{InLeapYear}(t) & \text{if MonthFromTime}(t)=11
\end{array}
$$

15.9.1.6 Week Day

The weekday for a particular time value t is defined as

$$\text{WeekDay}(t) = (\text{Day}(t) + 4) \text{ modulo } 7$$

A weekday value of 0 specifies Sunday; 1 specifies Monday; 2 specifies Tuesday; 3 specifies Wednesday; 4 specifies Thursday; 5 specifies Friday; and 6 specifies Saturday. Note that $\text{WeekDay}(0) = 4$, 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. The value LocalTZA does not vary with time but depends only on the geographic location.

15.9.1.8 Daylight Saving Time Adjustment

An implementation of ECMAScript is expected to determine the daylight saving time algorithm. The algorithm to determine the daylight saving time adjustment DaylightSavingTA(t), measured in milliseconds, must depend only on four things:

(1) the time since the beginning of the year

$$t - \text{TimeFromYear}(\text{YearFromTime}(t))$$

(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 not try to determine whether the exact time was subject to daylight saving time, but just whether daylight saving time would have been in effect if 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 saving time year round.

If the host environment provides functionality for determining daylight saving time, the implementation of ECMAScript is free to map the year in question to an equivalent year (same leap-year-ness and same starting week day for the year) for which the host environment provides daylight saving time information. The only restriction is that all equivalent years should produce the same result.

15.9.1.9 Local Time

Conversion from UTC to local time is defined by

LocalTime(*t*) = *t* + LocalTZA + DaylightSavingTA(*t*)

Conversion from local time to UTC is defined by

UTC(*t*) = *t* – LocalTZA – DaylightSavingTA(*t* – 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:

HourFromTime(*t*) = floor(*t* / msPerHour) modulo HoursPerDay

MinFromTime(*t*) = floor(*t* / msPerMinute) modulo MinutesPerHour

SecFromTime(*t*) = floor(*t* / msPerSecond) modulo SecondsPerMinute

msFromTime(*t*) = t modulo msPerSecond

where

HoursPerDay = 24

MinutesPerHour = 60

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*) **==** *ym* and MonthFromTime(*t*) **==** *mn* and DateFromTime(*t*) **==** 1; but if this is not possible (because some argument is out of range), return **NaN**.
8. Return Day(*t*) + *dt* – 1.

15.9.1.13 MakeDate (day, time)

The operator Make Date calculates a n umber 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 x 10^{15}, return **NaN**.
3. Return an implementation-dependent choice of either ToInteger(*time*) or ToInteger(*time*) + (**+0**). (Adding a positive zero converts **–0** to **+0**.)

NOTE The point of step 3 is that an implementation is permitted a choice of internal representations of time values, for example as a 64-bit signed integer or as a 64-bit floating-point value. Depending on the implementation, this internal representation may or may not distinguish **–0** and **+0**.

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**". The value of an absent time zone offset 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 Called as a Function

When **`Date`** is called as a function rather than as a constructor, it returns a String representing the current time (UTC).

NOTE The function call **`Date(…)`** is not equivalent to the object creation expression **`new Date(…)`** with the same arguments.

15.9.2.1 Date ([year [, month [, date [, hours [, minutes [, seconds [, ms]]]]]]])

All of the arguments are optional; any arguments supplied 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 the standard built-in method **`Date.prototype.toString`**.

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 object.

15.9.3.1 new Date (year, month [, date [, hours [, minutes [, seconds [, ms]]]]])

When **Date** is called with two to seven arguments, it computes the date from *year*, *month*, and (optionally) *date*, *hours*, *minutes*, *seconds* and *ms*.

The [[Prototype]] internal 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"`**.

The [[Extensible]] internal property of the newly constructed object is set to **true**.

The [[PrimitiveValue]] internal property of the newly constructed object is set as follows:

1. Let *y* be ToNumber(*year*).
2. Let *m* be ToNumber(*month*).
3. If *date* is supplied then let *dt* be ToNumber(*date*); else let *dt* be **1**.
4. If *hours* is supplied then let *h* be ToNumber(*hours*); else let *h* be **0**.
5. If *minutes* is supplied then let *min* be ToNumber(*minutes*); else let *min* be **0**.
6. If *seconds* is supplied then let *s* be ToNumber(*seconds*); else let *s* be **0**.
7. If *ms* is supplied then let *milli* be ToNumber(*ms*); else let *milli* be **0**.
8. If *y* is not **NaN** and 0 ≤ ToInteger(*y*) ≤ 99, then let *yr* be 1900+ToInteger(*y*); otherwise, let *yr* be *y*.
9. Let *finalDate* be MakeDate(MakeDay(*yr*, *m*, *dt*), MakeTime(*h*, *min*, *s*, *milli*)).
10. Set the [[PrimitiveValue]] internal property of the newly constructed object to TimeClip(UTC(*finalDate*)).

15.9.3.2 new Date (value)

The [[Prototype]] internal 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"`**.

The [[Extensible]] internal property of the newly constructed object is set to **true**.

The [[PrimitiveValue]] internal property of the newly constructed object is set as follows:

1. Let *v* be ToPrimitive(*value*).
2. If Type(*v*) is String, then

a. Parse *v* as a date, in exactly the same manner as for the **parse** method (15.9.4.2); let *V* be the time value for this date.

3. Else, let *V* be ToNumber(*v*).
4. Set the [[PrimitiveValue]] internal property of the newly constructed object to TimeClip(*V*) and return.

15.9.3.3 new Date ( )

The [[Prototype]] internal 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"**.

The [[Extensible]] internal property of the newly constructed object is set to **true**.

The [[PrimitiveValue]] internal property of the newly constructed object is set to the time value (UTC) identifying the current time.

15.9.4 Properties of the Date Constructor

The value of the [[Prototype]] internal property of the Date constructor is the Function prototype object (15.3.4).

Besides the internal properties and the **length** property (whose value is **7**), the Date constructor has the following properties:

15.9.4.1 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** }.

15.9.4.2 Date.parse (string)

The **parse** function applies the ToString operator to its argument and 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**.

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.

15.9.4.3 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. Let *m* be ToNumber(*month*).
3. If *date* is supplied then let *dt* be ToNumber(*date*); else let *dt* be **1**.
4. If *hours* is supplied then let *h* be ToNumber(*hours*); else let *h* be **0**.
5. If *minutes* is supplied then let *min* be ToNumber(*minutes*); else let *min* be **0**.
6. If *seconds* is supplied then let *s* be ToNumber(*seconds*); else let *s* be **0**.
7. If *ms* is supplied then let *milli* be ToNumber(*ms*); else let *milli* be **0**.
8. If *y* is not **NaN** and 0 ≤ ToInteger(*y*) ≤ 99, then let *yr* be 1900+ToInteger(*y*); otherwise, let *yr* be *y*.
9. Return TimeClip(MakeDate(MakeDay(*yr*, *m*, *dt*), MakeTime(*h*, *min*, *s*, *milli*))).

The **length** property of the **UTC** function is **7**.

NOTE The `UTC` function differs from the `Date` constructor in two ways: it returns a time value as a Number, rather than creating a Date object, and it interprets the arguments in UTC rather than as local time.

15.9.4.4 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**.

15.9.5 Properties of the Date Prototype Object

The Date prototype object is itself a Date object (its [[Class]] is **"Date"**) whose [[PrimitiveValue]] is **NaN**.

The value of the [[Prototype]] internal property of the Date prototype object is the standard built-in Object prototype object (15.2.4).

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. Unless explicitly noted otherwise, none of these functions are generic; a **TypeError** exception is thrown if the **this** value is not an object for which the value of the [[Class]] internal 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 [[PrimitiveValue]] internal property of this Date object.

15.9.5.1 Date.prototype.constructor

The initial value of **Date.prototype.constructor** is the built-in **Date** constructor.

15.9.5.2 Date.prototype.toString ( )

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.

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.

15.9.5.3 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.

15.9.5.4 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.

15.9.5.5 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.

15.9.5.6 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.

15.9.5.7 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.

15.9.5.8 Date.prototype.valueOf ( )

The **`valueOf`** function returns a Number, which is this time value.

15.9.5.9 Date.prototype.getTime ( )

1. Return this time value.

15.9.5.10 Date.prototype.getFullYear ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(LocalTime(*t*)).

15.9.5.11 Date.prototype.getUTCFullYear ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(*t*).

15.9.5.12 Date.prototype.getMonth ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MonthFromTime(LocalTime(*t*)).

15.9.5.13 Date.prototype.getUTCMonth ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MonthFromTime(*t*).

15.9.5.14 Date.prototype.getDate ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return DateFromTime(LocalTime(*t*)).

15.9.5.15 Date.prototype.getUTCDate ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return DateFromTime(*t*).

15.9.5.16 Date.prototype.getDay ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return WeekDay(LocalTime(*t*)).

15.9.5.17 Date.prototype.getUTCDay ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return WeekDay(*t*).

15.9.5.18 Date.prototype.getHours ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return HourFromTime(LocalTime(*t*)).

15.9.5.19 Date.prototype.getUTCHours ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return HourFromTime(*t*).

15.9.5.20 Date.prototype.getMinutes ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MinFromTime(LocalTime(*t*)).

15.9.5.21 Date.prototype.getUTCMinutes ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MinFromTime(*t*).

15.9.5.22 Date.prototype.getSeconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return SecFromTime(LocalTime(*t*)).

15.9.5.23 Date.prototype.getUTCSeconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return SecFromTime(*t*).

15.9.5.24 Date.prototype.getMilliseconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return msFromTime(LocalTime(*t*)).

15.9.5.25 Date.prototype.getUTCMilliseconds ( )

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return msFromTime(*t*).

15.9.5.26 Date.prototype.getTimezoneOffset ( )

Returns the difference between local time and UTC time in minutes.

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return (*t* – LocalTime(*t*)) / msPerMinute.

15.9.5.27 Date.prototype.setTime (time)

1. Let *v* be TimeClip(ToNumber(*time*)).
2. Set the [[PrimitiveValue]] internal property of this Date object to *v*.
3. Return *v*.

15.9.5.28 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 [[PrimitiveValue]] internal property of this Date object to *u*.

5. Return *u*.

15.9.5.29 Date.prototype.setUTCMilliseconds (ms)

1. Let *t* be this time value.
2. Let *time* be MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), ToNumber(*ms*)).
3. Let *v* be TimeClip(MakeDate(Day(*t*), *time*)).
4. Set the [[PrimitiveValue]] internal property of this Date object to *v*.
5. Return *v*.

15.9.5.30 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 [[PrimitiveValue]] internal property of this Date object to *u*.
7. Return *u*.

The **`length`** property of the **`setSeconds`** method is **2**.

15.9.5.31 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. 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 *v* be TimeClip(*date*).
6. Set the [[PrimitiveValue]] internal property of this Date object to *v*.
7. Return *v*.

The **`length`** property of the **`setUTCSeconds`** method is **2**.

15.9.5.32 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 [[PrimitiveValue]] internal property of this Date object to *u*.
8. Return *u*.

The **`length`** property of the **`setMinutes`** method is **3**.

15.9.5.33 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. 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 *v* be TimeClip(*date*).
7. Set the [[PrimitiveValue]] internal property of this Date object to *v*.
8. Return *v*.

The **`length`** property of the **`setUTCMinutes`** method is **3**.

15.9.5.34 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 [[PrimitiveValue]] internal property of this Date object to *u*.
9. Return *u*.

The **`length`** property of the **`setHours`** method is **4**.

15.9.5.35 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. 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 *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 *newDate* be MakeDate(Day(*t*), MakeTime(*h*, *m*, *s*, *milli*)).
7. Let *v* be TimeClip(*newDate*).
8. Set the [[PrimitiveValue]] internal property of this Date object to *v*.
9. Return *v*.

The **`length`** property of the **`setUTCHours`** method is **4**.

15.9.5.36 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 [[PrimitiveValue]] internal property of this Date object to *u*.
6. Return *u*.

15.9.5.37 Date.prototype.setUTCDate (date)

1. Let *t* be this time value.
2. Let *dt* be ToNumber(*date*).
3. Let *newDate* be MakeDate(MakeDay(YearFromTime(*t*), MonthFromTime(*t*), *dt*), TimeWithinDay(*t*)).
4. Let *v* be TimeClip(*newDate*).
5. Set the [[PrimitiveValue]] internal property of this Date object to *v*.
6. Return *v*.

15.9.5.38 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 [[PrimitiveValue]] internal property of this Date object to *u*.
7. Return *u*.

The **`length`** property of the **`setMonth`** method is **2**.

15.9.5.39 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. 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 *v* be TimeClip(*newDate*).
6. Set the [[PrimitiveValue]] internal property of this Date object to *v*.
7. Return *v*.

The **`length`** property of the **`setUTCMonth`** method is **2**.

15.9.5.40 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 [[PrimitiveValue]] internal property of this Date object to *u*.
8. Return *u*.

The **`length`** property of the **`setFullYear`** method is **3**.

15.9.5.41 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. 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 *v* be TimeClip(*newDate*).
7. Set the [[PrimitiveValue]] internal property of this Date object to *v*.
8. Return *v*.

The **`length`** property of the **`setUTCFullYear`** method is **3**.

15.9.5.42 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.

15.9.5.43 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.

15.9.5.44 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 calling the [[Get]] internal method of *O* with argument "**`toISOString`**".
5. If IsCallable(*toISO*) is **false**, throw a **TypeError** exception.
6. Return the result of calling the [[Call]] internal method of *toISO* with *O* as the **this** value and an empty argument list.

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.

15.9.6 Properties of Date Instances

Date instances inherit properties from the Date prototype object and their [[Class]] internal property value is **`"Date"`**. Date instances also have a [[PrimitiveValue]] internal property.

The [[PrimitiveValue]] internal property is 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*

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 [[Match]] internal property.

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 the setting of the RegExp object's **`ignoreCase`** property.
- *Multiline* is the setting of the RegExp object's **`multiline`** property.

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.

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,aaaaaaaaaaaaaa".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**.

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.

The abstract operation *IsWordChar* takes an integer parameter *e* and performs the following:

1. If *e* == –1 or *e* == *InputLength*, return **false**.
2. Let *c* be the character *Input*[*e*].
3. If *c* is one of the sixty-three characters below, return **true**.

    a b c d e f g h i j k l m n o p q r s t u v w x y z
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
    0 1 2 3 4 5 6 7 8 9 _
4. Return **false**.

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.

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* == *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*) == *cc*, return **failure**.
    6. Else *invert* is **true**,
        a. If there exists a member *a* of set *A* such that *Canonicalize*(*a*) == *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.

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.
    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.