Standard ECMA-262
2nd Edition - August 1998

ECMA

Standardizing Information and Communication Systems

ECMAScript Language Specification

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

Standard ECMA-262

2nd Edition - August 1998

ECMA

Standardizing Information and Communication Systems

ECMAScript Language Specification

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

MB E-262-ii.doc 31-08-98 17,06

Brief History

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

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

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

The work on standardization of the language continues to support regular expressions, richer control statements and better string handling, in addition to the core language standardized in the first two editions of the ECMA Standard. These features and others, such as try/catch exception handling and better internationalization facilities, are being documented in anticipation of the third edition of the standard about the end of 1999 which will contain the second version of the language.

This Standard has been adopted as 2nd Edition of ECMA-262 by the ECMA General Assembly in August 1998.

Table of contents

1 Scope

This Standard defines the ECMAScript scripting language.

2 Conformance

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

A conforming implementation of this International standard shall interpret characters in conformance with the Unicode Standard, Version 2.0, and ISO/IEC 10646-1 with UCS-2 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300.

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

A conforming implementation of ECMAScript is permitted to support program syntax not described in this specification. In particular, a conforming implementation of ECMAScript is permitted to support program syntax that makes use of the "future reserved words" listed in section Future Reserved Word of this specification.

3 References

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

ISO/IEC 10646-1:1993 Information Technology -- Universal Multiple-Octet Coded Character Set (UCS), including amendments 1 through 9 and technical corrigendum 1.

ISO/IEC 646.IRV:1991 -- Information Processing -- ISO 7-bit Coded Character Set for Information Interchange.

Unicode Inc. (1996), The Unicode Standard™, Version 2.0. ISBN: 0-201-48345-9, Addison-Wesley Publishing Co., Menlo Park, California.

ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, New York (1985).

4 Overview

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

ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment. ECMAScript as defined here is not intended to be computationally self-sufficient; indeed, there are no provisions in this specification for input of external data or output of computed results. Instead, it is expected that the computational environment of an ECMAScript program will provide not only the objects and other facilities described in this specification but also certain environment-specific *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, and therefore there may be a number of informalities built into the language.

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

ECMAScript 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™ and Self, 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.

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 an unordered collection of ***properties*** each with 0 or more ***attributes*** which determine how each property can be used—for example, when the ReadOnly attribute for a property is set to true, any attempt by executed ECMAScript code to change the value of the property has no effect. Properties are containers that hold other objects, ***primitive values***, or ***methods***. 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 method is a function associated with an object via a property.

ECMAScript defines a collection of ***built-in objects*** which 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, and the **Date** object.

ECMAScript also defines a set of built-in ***operators*** that may not be, strictly speaking, functions or methods. 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.

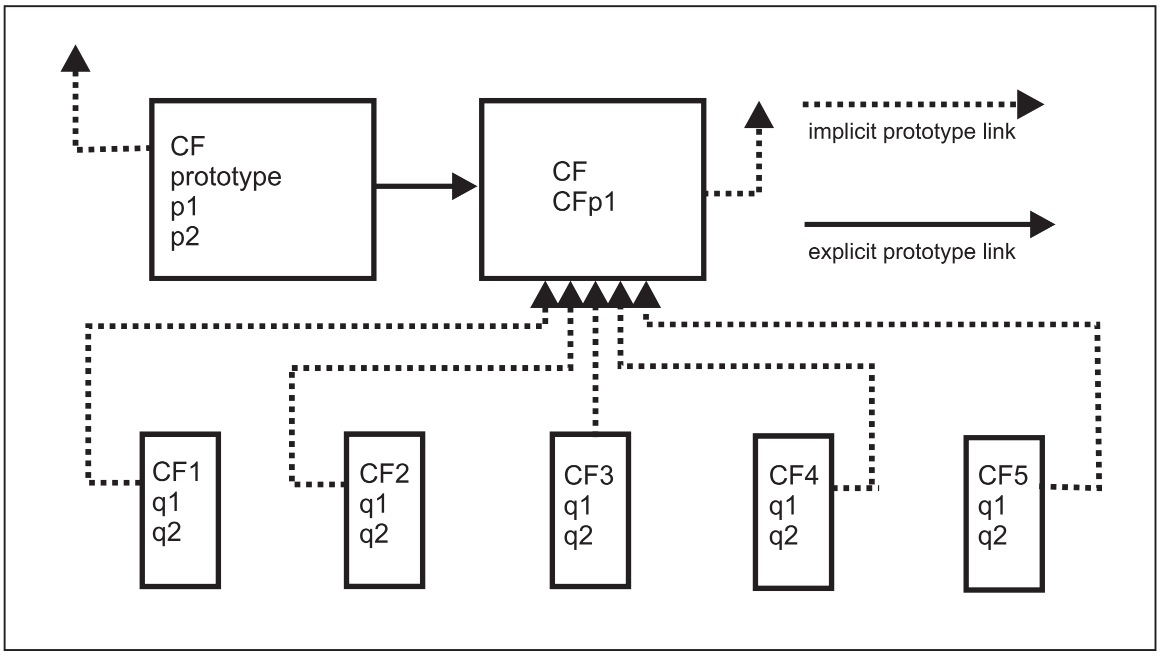
4.2.1 Objects

ECMAScript does not contain proper classes such as those in C++, Smalltalk, or Java, but rather, supports ***constructors*** which create objects by executing code that allocates storage for the objects and initialises all or part of them by assigning initial values to their properties. All functions including constructors are objects, but not all objects are constructors. Each constructor has a **Prototype** property that is used to implement ***prototype-based inheritance*** and ***shared properties***. Objects are created by using constructors in **new** expressions; for example, `new String("A String")` creates a new string object. Invoking a constructor without using **new** has consequences that depend on the constructor. For example, `String("A String")` produces a primitive string, not an object.

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

In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behaviour. In ECMAScript, the state and methods are carried by objects, 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. The following diagram illustrates this:



CF is a constructor (and also an object). Five objects have been created by using new expressions: CF1, CF2, CF3, CF4, and CF5. Each of these objects contains properties named q1 and q2. The dashed lines represent the implicit prototype relationship; so, for example, CF3's prototype is CFp. The constructor, CF, has two properties itself, named p1 and p2, which are not visible to CFp, CF1, CF2, CF3, CF4, or CF5. The property named CFp1 in CFp is shared by CF1, CF2, CF3, CF4, and CF5, as are any properties found in CFp's implicit prototype chain which are not named q1, q2, or CFp1. Notice that there is no implicit prototype link between CFp and CF.

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 CF1, CF2, CF3, CF4, and CF5 by assigning a new value to the property in CFp.

4.3 Definitions

The following are informal definitions of key terms associated with ECMAScript.

4.3.1 Type

A ***type*** is a set of data values.

4.3.2 Primitive value

A ***primitive value*** is a member of one of the types **Undefined**, **Null**, **Boolean**, **Number**, or **String**. A primitive value is a datum that is represented directly at the lowest level of the language implementation.

4.3.3 Object

An ***object*** is a member of the type **Object**. It is an unordered collection of properties each of which contains a primitive value, object, or function. A function stored in a property of an object is called a method.

4.3.4 Constructor

A ***constructor*** is a function object that creates and initialises objects. Each constructor has an associated prototype object that is used to implement inheritance and shared properties.

4.3.5 Prototype

A ***prototype*** is an object used to implement structure, state, and behaviour inheritance in ECMAScript. When a constructor creates an object, that object implicitly references the constructor's associated prototype for the purpose of resolving property references. The constructor's associated prototype 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.

4.3.6 Native object

A ***native object*** is any object supplied by an ECMAScript implementation independent of the host environment. 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

A ***built-in object*** is any object supplied by an ECMAScript implementation, independent of the host environment, which is present at the start of the execution of an ECMAScript program. Standard built-in objects are defined in this specification, and the ECMAScript implementation may specify and define others. Every built-in object is a native object.

4.3.8 Host object

A ***host object*** is any object supplied by the host environment to complete the execution environment of ECMAScript. Any object that is not native is a host object.

4.3.9 Undefined value

The ***undefined value*** is a primitive value used when a variable has not been assigned a value.

4.3.10 Undefined type

The type **Undefined** has exactly one value, called **undefined**.

4.3.11 Null value

The ***null value*** is a primitive value that represents the null, empty, or non-existent reference.

4.3.12 Null type

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

4.3.13 Boolean value

A ***boolean value*** is a member of the type **Boolean** and is one of two unique values, **true** and **false**.

4.3.14 Boolean type

The type **Boolean** represents a logical entity and consists of exactly two unique values. One is called **true** and the other is called **false**.

4.3.15 Boolean object

A ***boolean object*** is a member of the type **Object** and is an instance of the built-in Boolean object. That is, a boolean object is created by using the Boolean constructor in a new expression, supplying a boolean as an argument. The resulting object has an implicit (unnamed) property that is the boolean. A boolean object can be coerced to a boolean value. A boolean object can be used anywhere a boolean value is expected.

This is an example of one of the conveniences built into ECMAScript—in this case, the purpose is to accommodate programmers of varying backgrounds. Those familiar with imperative or procedural programming languages may find boolean, string and number values more natural, while those familiar with object-oriented languages may find boolean, string and number objects more intuitive.

4.3.16 String value

A ***string value*** is a member of the type **String** and is a finite ordered sequence of zero or more Unicode characters.

4.3.17 String type

The type **String** is the set of all finite ordered sequences of zero or more Unicode characters.

4.3.18 String object

A ***string object*** is a member of the type **Object** and is an instance of the built-in String object. That is, a string object is created by using the String constructor in a new expression, supplying a string as an argument. The resulting object has an implicit (unnamed) property that is the string. A string object can be coerced to a string value. A string object can be used anywhere a string value is expected.

4.3.19 Number value

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

4.3.20 Number type

The type **Number** is a set of values representing numbers. In ECMAScript the set of values represent the double-precision 64-bit format IEEE 754 values including the special “Not-a-Number” (NaN) values, positive infinity, and negative infinity.

4.3.21 Number object

A ***number object*** is a member of the type **Object** and is an instance of the built-in Number object. That is, a number object is created by using the Number constructor in a new expression, supplying a number as an argument. The resulting object has an implicit (unnamed) property that is the number. A number object can be coerced to a number value. A number object can be used anywhere a number value is expected. Note that a number object can have shared properties by adding them to the Number prototype.

4.3.22 Infinity

The primitive value **Infinity** represents the positive infinite number value.

4.3.23 NaN

The primitive value **NaN** represents the set of IEEE Standard “Not-a-Number” values.

5 Notational Conventions

5.1 Syntactic and Lexical Grammars

This section describes the context-free grammars used in this specification to define the lexical and syntactic structure of an ECMAScript program.

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 one 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 grammar

A *lexical grammar* for ECMAScript is given in Section 7. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *InputElement*, that describe how sequences of Unicode 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 (see section7.8). Simple white space and single-line comments are simply 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 multi-line comment 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.

Productions of the lexical grammar are distinguished by having two colons "`::`" as separating punctuation.

5.1.3 The numeric string grammar

A second 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 the characters of the Unicode character set. This grammar appears in section 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 sections11, 12, 13, and 14. This grammar has ECMAScript tokens defined by the lexical grammar as its terminal symbols (see section 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 Unicode 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 syntax 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 sections11, 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 Grammar Notation

Terminal symbols of the lexical and string grammars, and some of the terminal symbols of the syntactic grammar, 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.

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:

WithStatement **:**

with (*Expression* **)** *Statement*

states that the nonterminal *WithStatement* represents the token **`with`**, 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 $Initializer_{opt}$

is a convenient abbreviation for:

VariableDeclaration **:**

Identifier
Identifier Initializer

and that:

IterationStatement **:**

`for (` $Expression_{opt}$ **`;`** $Expression_{opt}$ **`;`** $Expression_{opt}$ **`)`** *Statement*

is a convenient abbreviation for:

IterationStatement **:**

`for ( ;` $Expression_{opt}$ **`;`** $Expression_{opt}$ **`)`** *Statement*
`for (` *Expression* **`;`** $Expression_{opt}$ **`;`** $Expression_{opt}$ **`)`** *Statement*

which in turn is an abbreviation for:

IterationStatement **:**

`for ( ; ;` $Expression_{opt}$ **`)`** *Statement*
`for ( ;` *Expression* **`;`** $Expression_{opt}$ **`)`** *Statement*
`for (` *Expression* **`; ;`** $Expression_{opt}$ **`)`** *Statement*
`for (` *Expression* **`;`** *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 (`** *Expression* **`; ; )`** *Statement*
> **`for (`** *Expression* **`; ;`** *Expression* **`)`** *Statement*
> **`for (`** *Expression* **`;`** *Expression* **`; )`** *Statement*
> **`for (`** *Expression* **`;`** *Expression* **`;`** *Expression* **`)`** *Statement*

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

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:

ReturnStatement **:**

> **`return`** [no *LineTerminator* here] $Expression_{opt}$ **`;`**

indicates that the production may not be used if a *LineTerminator* occurs in the program between the **`return`** 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 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:

ZeroToThree **:: one of**

> **`0          1          2          3`**

which is merely a convenient abbreviation for:

ZeroToThree **::**

> **0**
> **1**
> **2**
> **3**

When an alternative in a production of the lexical grammar or the numeric string grammar appears to be a multicharacter 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 roman type in cases where it would be impractical to list all the alternatives:

SourceCharacter **::**

> *any Unicode character*

5.2 Algorithm Conventions

The specification often uses a numbered list to specify steps in an algorithm. These algorithms are used to clarify semantics. In practice, there may be more efficient algorithms available to implement a given feature.

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". Type(x) is used as shorthand for "the type of x".

Mathematical operations such as addition, subtraction, negation, multiplication, division, and the mathematical functions defined later in this section 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 such that abs(k) < abs(y) and $x-k = q \cdot y$ 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 "generate a runtime error", execution of the algorithm (and any calling algorithms) is terminated and no result is returned.

6 Source Text

ECMAScript source text is represented as a sequence of characters representable using the Unicode version 2.0 character encoding.

SourceCharacter **::**

any Unicode character

Except within comments and string literals, every ECMAScript program shall consist of only characters from the first 128 Unicode characters (that is, the first half of row zero). Other Unicode characters may appear only within comments and string literals. In string literals, any Unicode character may also be expressed as a Unicode escape sequence consisting of six characters from the first 128 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, the Unicode escape sequence contributes one character to the string value of the literal.

Although the characters in an ECMAScript program are Unicode characters, they are treated as independent 16-bit values with none of the context-dependent interpretation specified in the Unicode standard. Such values are often called "code points". The Unicode standard refers to code points as "coded character data elements". Throughout this International standard the terms "character" and "code point" are understood to mean "coded character data element".

NOTE ECMAScript differs from the Java programming language in the behaviour of Unicode escape sequences. In a Java program, if the Unicode escape sequence **\u000A**, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character **000A** is line feed) and therefore the next 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 either 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.

Syntax

InputElement **::**

WhiteSpace
LineTerminator
Comment
Token

7.1 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 may occur between any two tokens, and may occur within strings (where they are considered significant characters forming part of the literal string value), but cannot appear within any other kind of token.

The following characters are considered to be white space:

Unicode Value	*Name*	*Formal Name*
\u0009	Tab	<TAB>
\u000B	Vertical Tab	<VT>
\u000C	Form Feed	<FF>
\u0020	Space	<SP>

Syntax

WhiteSpace **::**

<TAB>
<VT>
<FF>
<SP>

7.2 Line Terminators

Like whitespace characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike whitespace 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. A line terminator cannot occur within any token, not even a string. Line terminators also affect the process of automatic semicolon insertion (see section 7.8).

The following characters are considered to be line terminators:

Unicode Value	*Name*	*Formal Name*
\u000A	Line Feed	<LF>
\u000D	Carriage Return	<CR>

Syntax

LineTerminator **::**

<LF>
<CR>

7.3 Comments

Description

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 section7.8.2).

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** *asterisk* `*`

MultiLineNotForwardSlashOrAsteriskChar **::**

SourceCharacter **but neither** *forward-slash* `/` **nor** *asterisk* `*`

SingleLineComment **::**

`//` *SingleLineCommentChars*$_{opt}$

SingleLineCommentChars **::**

> *SingleLineCommentChar SingleLineCommentChars*$_{opt}$

SingleLineCommentChar **::**

> *SourceCharacter* **but not** *LineTerminator*

7.4 Tokens

Syntax

Token **::**

> *ReservedWord*
> *Identifier*
> *Punctuator*
> *NumericLiteral*
> *StringLiteral*

7.4.1 Reserved Words

Description

Reserved words cannot be used as identifiers.

Syntax

ReservedWord **::**

> *Keyword*
> *FutureReservedWord*
> *NullLiteral*
> *BooleanLiteral*

7.4.2 Keywords

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

Syntax

Keyword :: one of

`break`	**`for`**	**`new`**	**`var`**
`continue`	**`function`**	**`return`**	**`void`**
`delete`	**`if`**	**`this`**	**`while`**
`else`	**`in`**	**`typeof`**	**`with`**

7.4.3 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**

`abstract`	**`do`**	**`import`**	**`short`**
`boolean`	**`double`**	**`instanceof`**	**`static`**
`byte`	**`enum`**	**`int`**	**`super`**
`case`	**`export`**	**`interface`**	**`switch`**
`catch`	**`extends`**	**`long`**	**`synchronized`**
`char`	**`final`**	**`native`**	**`throw`**
`class`	**`finally`**	**`package`**	**`throws`**

```
const           float           private         transient
debugger        goto            protected       try
default         implements      public          volatile
```

7.5 Identifiers

Description

An identifier is a character sequence of unlimited length, where each character in the sequence must be a letter, a decimal digit, an underscore (_) character, or a dollar sign (**$**) character, and the first character may not be a decimal digit. ECMAScript identifiers are case sensitive: identifiers whose characters differ in any way, even if only in case, are considered to be distinct. The dollar sign (**$**) character is intended for use only in mechanically generated code.

Syntax

Identifier **::**

IdentifierName **but not** *ReservedWord*

IdentifierName **::**

IdentifierLetter
IdentifierName IdentifierLetter
IdentifierName DecimalDigit

IdentifierLetter **:: 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
$ _
```

DecimalDigit **:: one of**

```
0 1 2 3 4 5 6 7 8 9
```

7.6 Punctuators

Syntax

Punctuator **:: one of**

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

7.7 Literals

Syntax

Literal **::**

NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral

7.7.1 Null Literals

Syntax

NullLiteral **::**

`null`

Semantics

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

7.7.2 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**.

7.7.3 Numeric Literals

Syntax

NumericLiteral **::**

DecimalLiteral
HexIntegerLiteral
OctalIntegerLiteral

DecimalLiteral **::**

DecimalIntegerLiteral **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits ExponentPart*$_{opt}$
DecimalIntegerLiteral ExponentPart$_{opt}$

DecimalIntegerLiteral **::**

0
NonZeroDigit DecimalDigits$_{opt}$

DecimalDigits **::**

DecimalDigit
DecimalDigits DecimalDigit

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`

OctalIntegerLiteral **::**

`0` *OctalDigit*
OctalIntegerLiteral OctalDigit

OctalDigit **:: one of**

`0 1 2 3 4 5 6 7`

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, if this mathematical value is not representable using the number type, it is rounded to either the nearest representable value type above the mathematical value or the nearest representable value below the mathematical value.

The rounding mechanism is unspecified, but implementations are encouraged to use IEEE 754 round-to-nearest.

- The MV of *NumericLiteral* **::** *DecimalLiteral* is the MV of *DecimalLiteral*.
- The MV of *NumericLiteral* **::** *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *NumericLiteral* **::** *OctalIntegerLiteral* is the MV of *OctalIntegerLiteral*.
- 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 *DecimalDigit*s.
- 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 *DecimalDigit*s 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 *DecimalDigit*s.

- The MV of *DecimalLiteral* **::** **.** *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times 10^{e-n}, where *n* is the number of characters in *DecimalDigit*s 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 DecimalDigits* is (the MV of *NonZeroDigit* times 10^n) plus the MV of *DecimalDigits*, where *n* is the number of characters in *DecimalDigits*.
- The MV of *DecimalDigits* **::** *DecimalDigit* is the MV of *DecimalDigit*.
- The MV of *DecimalDigits* **::** *DecimalDigits DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart* **::** *ExponentIndicator SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger* **::** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* **::** **+** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* **::** **-** *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* **::** **0** or of *HexDigit* **::** **0** or of *OctalDigit* **::** **0** is 0.
- The MV of *DecimalDigit* **::** **1** or of *NonZeroDigit* **::** **1** or of *HexDigit* **::** **1** or of *OctalDigit* **::** **1** is 1.
- The MV of *DecimalDigit* **::** **2** or of *NonZeroDigit* **::** **2** or of *HexDigit* **::** **2** or of *OctalDigit* **::** **2** is 2.
- The MV of *DecimalDigit* **::** **3** or of *NonZeroDigit* **::** **3** or of *HexDigit* **::** **3** or of *OctalDigit* **::** **3** is 3.
- The MV of *DecimalDigit* **::** **4** or of *NonZeroDigit* **::** **4** or of *HexDigit* **::** **4** or of *OctalDigit* **::** **4** is 4.
- The MV of *DecimalDigit* **::** **5** or of *NonZeroDigit* **::** **5** or of *HexDigit* **::** **5** or of *OctalDigit* **::** **5** is 5.
- The MV of *DecimalDigit* **::** **6** or of *NonZeroDigit* **::** **6** or of *HexDigit* **::** **6** or of *OctalDigit* **::** **6** is 6.
- The MV of *DecimalDigit* **::** **7** or of *NonZeroDigit* **::** **7** or of *HexDigit* **::** **7** or of *OctalDigit* **::** **7** is 7.
- The MV of *DecimalDigit* **::** **8** or of *NonZeroDigit* **::** **8** or of *HexDigit* **::** **8** is 8.
- The MV of *DecimalDigit* **::** **9** or of *NonZeroDigit* **::** **9** or of *HexDigit* **::** **9** is 9.
- The MV of *HexDigit* **::** **a** or of *HexDigit* **::** **A** is 10.
- The MV of *HexDigit* **::** **b** or of *HexDigit* **::** **B** is 11.
- The MV of *HexDigit* **::** **c** or of *HexDigit* **::** **C** is 12.
- The MV of *HexDigit* **::** **d** or of *HexDigit* **::** **D** is 13.
- The MV of *HexDigit* **::** **e** or of *HexDigit* **::** **E** is 14.
- The MV of *HexDigit* **::** **f** or of *HexDigit* **::** **F** is 15.
- The MV of *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*.
- The MV of *OctalIntegerLiteral* **::** **0** *OctalDigit* is the MV of *OctalDigit*.

- The MV of *OctalIntegerLiteral* **::** *OctalIntegerLiteral OctalDigit* is (the MV of *OctalIntegerLiteral* times 8) plus the MV of *OctalDigit*.

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 (in the sense defined in section8.5), unless the literal is a *DecimalLiteral* and the literal has more than 20 significant digits, in which case the number value may be either the number value for the MV of a literal produced by replacing each significant digit after the 20th with a **0** digit or the number value for the MV of a literal produced by replacing each significant digit after the 20th with a **0** digit and then incrementing the literal at the 20th significant digit position. A digit is *significant* if it is not part of an *ExponentPart* and

- it is not **0;** or
- there is a nonzero digit to its left and there is a nonzero digit, not in the *ExponentPart*, to its right.

7.7.4 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.

Syntax

StringLiteral **::**

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

DoubleStringCharacters **::**

    DoubleStringCharacter DoubleStringCharacters$_{opt}$

SingleStringCharacters **::**

    SingleStringCharacter SingleStringCharacters$_{opt}$

DoubleStringCharacter **::**

    SourceCharacter **but not** *double-quote* **"** **or** *backslash* \ **or** *LineTerminator*
    EscapeSequence

SingleStringCharacter **::**

    SourceCharacter **but not** *single-quote* **'** **or** *backslash* \ **or** *LineTerminator*
    EscapeSequence

EscapeSequence **::**

    CharacterEscapeSequence
    OctalEscapeSequence
    HexEscapeSequence
    UnicodeEscapeSequence

CharacterEscapeSequence **::**

    \ *SingleEscapeCharacter*
    \ *NonEscapeCharacter*

SingleEscapeCharacter **::** **one of**

```
    '    "    \    b    f    n    r    t
```

NonEscapeCharacter **::**

SourceCharacter **but not** *EscapeCharacter* **or** *LineTerminator*

EscapeCharacter **::**

SingleEscapeCharacter
OctalDigit
`x`
`u`

HexEscapeSequence **::**

`\x` *HexDigit HexDigit*

OctalEscapeSequence **::**

\ *OctalDigit*
\ *OctalDigit OctalDigit*
\ *ZeroToThree OctalDigit OctalDigit*

ZeroToThree **:: one of**

`0        1        2        3`

UnicodeEscapeSequence **::**

`\u` *HexDigit HexDigit HexDigit HexDigit*

The definitions of the nonterminals *HexDigit* and *OctalDigit* are given in section 7.7.3. *SourceCharacter* is described in sections 2 and 6.

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 section 7.7.3.

- 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 CV of *DoubleStringCharacter* **::** *SourceCharacter* **but not** *double-quote* **`"`** **or** *backslash* \ **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *DoubleStringCharacter* **::** *EscapeSequence* is the CV of the *EscapeSequence*.

- The CV of *SingleStringCharacter* **::** *SourceCharacter* **but not** *single-quote* **`'`** **or** *backslash* **`\`** **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *SingleStringCharacter* **::** *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *EscapeSequence* **::** *CharacterEscapeSequence* is the CV of the *CharacterEscapeSequence*.
- The CV of *EscapeSequence* **::** *OctalEscapeSequence* is the CV of the *OctalEscapeSequence*.
- 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 Unicode character whose Unicode value is determined by the *SingleEscapeCharacter* according to the following table:

Escape Sequence	***Unicode Value***	***Name***	***Symbol***
`\b`	**`\u0008`**	backspace	<BS>
`\t`	**`\u0009`**	horizontal tab	<HT>
`\n`	**`\u000A`**	line feed (new line)	<LF>
`\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** *EscapeCharacter* **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *HexEscapeSequence* **::** **`\x`** *HexDigit HexDigit* is the Unicode character whose code is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.
- The CV of *OctalEscapeSequence* **::** **`\`** *OctalDigit* is the Unicode character whose code is the MV of the *OctalDigit*.
- The CV of *OctalEscapeSequence* **::** **`\`** *OctalDigit OctalDigit* is the Unicode character whose code is (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The CV of *OctalEscapeSequence* **::** **`\`** *ZeroToThree OctalDigit OctalDigit* is the Unicode character whose code is (64 (that is, 8^2) times the MV of the *ZeroToThree*) plus (8 times the MV of the first *OctalDigit*) plus the MV of the second *OctalDigit*.
- The MV of *ZeroToThree* **::** **`0`** is 0.
- The MV of *ZeroToThree* **::** **`1`** is 1.
- The MV of *ZeroToThree* **::** **`2`** is 2.
- The MV of *ZeroToThree* **::** **`3`** is 3.
- The CV of *UnicodeEscapeSequence* **::** **`\u`** *HexDigit HexDigit HexDigit HexDigit* is the Unicode character whose code is (4096 (that is, 16^3) times the MV of the first *HexDigit*) plus (256 (that is, 16^2) times the MV of the second *HexDigit*) plus (16 times the MV of the third *HexDigit*) plus the MV of the fourth *HexDigit*.

NOTE A *LineTerminator* character cannot appear in a string literal, even if preceded by a backslash `\`. 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 Automatic semicolon insertion

Certain ECMAScript statements (empty statement, variable statement, expression statement, **`continue`** statement, **`break`** statement, and **`return`** statement) must each be terminated with a semicolon. Such a semicolon 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.

7.8.1 Rules of automatic semicolon insertion

- 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 and the parser is not currently parsing the header of a **`for`** statement, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:

  1. The offending token is separated from the previous token by at least one *LineTerminator*.
  2. The offending token is **`}`**.

- 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.

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.

- 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 there are two cases:

  1. If the parser is not currently parsing the header of a **`for`** statement, a semicolon is automatically inserted before the restricted token.
  2. If the parser is currently parsing the header of a **`for`** statement, it is a syntax error.

  These are the only restricted productions in the grammar:

PostfixExpression **:**

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

ReturnStatement **:**

`return` [no *LineTerminator* here] *Expression*$_{opt}$ **`;`**

The practical effect of these restricted productions is as follows:

1. When the token **`++`** or **`--`** 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.
2. When the token **`return`** is encountered and a *LineTerminator* is encountered before the next token is encountered, a semicolon is automatically inserted after the token **`return`**.

The resulting practical advice to ECMAScript programmers is:

1. A postfix **`++`** or **`--`** operator should appear on the same line as its operand.
2. An *Expression* in a **`return`** statement should start on the same line as the **`return`** token.

7.8.2 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 place where a semicolon is needed is within the header of a **`for`** statement. Automatic semicolon insertion never occurs within 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

A value is an entity that takes on one of nine types. There are nine types (**Undefined**, **Null**, **Boolean**, **String**, **Number**, **Object**, **Reference**, **List,** and **Completion)**. Values of type **Reference**, **List,** and **Completion** are used only as intermediate results of expression evaluation and cannot be stored to properties of objects.

8.1 The Undefined type

The Undefined type has exactly one value, called **undefined**. Any variable that has not been assigned a value is of type 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 Unicode characters (more properly referred to as code points; see section 6). Each character is regarded as occupying a position within the sequence. These positions are identified by nonnegative integers. The leftmost character (if any) is at position 0, the next character (if any) at position 1, and so on. The length of a string is the number of distinct positions within it. The empty string has length zero and therefore contains no characters.

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`**, assuming that the globally defined variable **`NaN`** has not been altered by program execution.) In some implementations, external code might be able to detect a difference between various Non-a-Number values, but such behaviour is implementation-dependent; to ECMAScript code, all **NaN** values are indistinguishable from each other.

There are two other special values, called **positive Infinity** and **negative Infinity**. For brevity, these values are also referred to for expository purposes by the symbols $+\infty$ and $-\infty$, respectively. (Note that these two infinite number values are produced by the program expressions **`+Infinity`** (or simply **`Infinity`**) and **`-Infinity`**, assuming that the globally defined variable **`Infinity`** has not been altered by program execution.)

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 there is a corresponding negative number 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 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

$$s \cdot m \cdot 2^{e}$$

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 denormalized, having the form

$$s \cdot m \cdot 2^{e}$$

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 $+1 \cdot 2^{53} \cdot 2^{971}$) and -2^{1024} (which is $-1 \cdot 2^{53} \cdot 2^{971}$). 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 sections 9.5 and 9.6, respectively.

8.6 The Object type

An Object is an unordered collection of properties. Each property consists of a name, a value and a set of attributes.

8.6.1 Property attributes

A property can have zero or more attributes from the following set:

Attribute	*Description*
ReadOnly	The property is a read-only property. Attempts by ECMAScript code to write to the property will be ignored. (Note, however, that in some cases the value of a property with the ReadOnly attribute may change over time because of actions taken by the underlying implementation; therefore "ReadOnly" does not mean "constant and unchanging"!).
DontEnum	The property is not to be enumerated by a **`for-in`** enumeration (section 12.6.3).
DontDelete	Attempts to delete the property will be ignored. See the description of the **`delete`** operator in section 11.4.1.
Internal	Internal properties have no name and are not directly accessible via the property accessor operators. How these properties are accessed is implementation specific. How and when some of these properties are used is specified by the language specification.

8.6.2 Internal Properties and Methods

Internal properties and methods are not exposed in the language. For the purposes of this document, their names 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 runtime error is generated.

There are two types of access for exposed properties: *get* and *put*, corresponding to retrieval and assignment, respectively.

Native ECMAScript objects have an internal property called [[Prototype]]. The value of this property is either **`null`** or an object and is used for implementing inheritance. Properties of the [[Prototype]] object are exposed as properties of the child object for the purposes of get access, but not for put access.

The following table summarises the internal properties used by this specification. The description indicates their behaviour for native ECMAScript objects. Host objects may implement these internal methods with any implementation-dependent behaviour, or it may be that a host object implements only some internal methods and not others.

Property	***Parameters***	***Description***
[[Prototype]]	none	The prototype of this object.
[[Class]]	none	A string value indicating the kind of this object.
[[Value]]	none	Internal state information associated with this object.
[[Get]]	(PropertyName)	Returns the value of the property.
[[Put]]	(PropertyName, Value)	Sets the specified property to Value.
[[CanPut]]	(PropertyName)	Returns a boolean value indicating whether a [[Put]] operation with the specified PropertyName will succeed.
[[HasProperty]]	(PropertyName)	Returns a boolean value indicating whether the object already has a member with the given name.
[[Delete]]	(PropertyName)	Removes the specified property from the object.
[[DefaultValue]]	(Hint)	Returns a default value for the object, which should be a primitive value (not an object or reference).
[[Construct]]	a list of argument values provided by the caller	Constructs an object. Invoked via the **`new`** operator. Objects that implement this internal method are called *constructors*.
[[Call]]	a list of argument values provided by the caller	Executes code associated with the object. Invoked via a function call expression. Objects that implement this internal method are called *functions*.

Every object must implement the [[Class]] property and the [[Get]], [[Put]], [[HasProperty]], [[Delete]], and [[DefaultValue]] methods, even host objects. (Note, however, that the [[DefaultValue]] method may, for some objects, simply generate a runtime error.)

The value of the [[Prototype]] property must be either an object or **null**, and every [[Prototype]] chain must have finite length (that is, starting from any object, recursively accessing the [[Prototype]] property must eventually lead to a **null** value). Whether or not a native object can have a host object as its [[Prototype]] depends on the implementation.

The value of the [[Class]] property is defined by this specification for every kind of built-in object. The value of the [[Class]] property of a host object may be any value, even a value used by a built-in object for its [[Class]] property. Note that this specification does not provide any means for a program to access the value of a [[Class]] property; that value is used internally to distinguish different kinds of built-in objects.

Every native object implements the [[Get]], [[Put]], [[CanPut]], [[HasProperty]], and [[Delete]] methods in the manner described in sections 8.6.2.1, 8.6.2.2, 8.6.2.3, 8.6.3.4, and 8.6.2.5, respectively, except that Array objects have a slightly different implementation of the [[Put]] method (section 15.4.5.1). Host objects may implement these methods in any manner; 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**.

In the following algorithm descriptions, assume *O* is a native ECMAScript object and *P* is a string.

8.6.2.1 [[Get]](P)

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

1. If *O* doesn't have a property with name *P*, go to step 4.
2. Get the value of the property.
3. Return Result(2).
4. If the [[Prototype]] of *O* is **null**, return **undefined**.
5. Call the [[Get]] method of [[Prototype]] with property name *P*.
6. Return Result(5).

8.6.2.2 [[Put]](P, V)

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

1. Call the [[CanPut]] method of *O* with name *P*.
2. If Result(1) is false, return.
3. If *O* doesn't have a property with name *P*, go to step 6.
4. Set the value of the property to *V*. The attributes of the property are not changed.
5. Return.
6. Create a property with name *P*, set its value to *V* and give it empty attributes.
7. Return.

Note, however, that if *O* is an Array object, it has a more elaborate [[Put]] method (section 15.4.5.1).

8.6.2.3 [[CanPut]](P)

The [[CanPut]] method is used only by the [[Put]] method.

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

1. If *O* doesn't have a property with name *P*, go to step 4.
2. If the property has the ReadOnly attribute, return **false**.
3. Return **true**.
4. If the [[Prototype]] of *O* is **null**, return **true**.
5. If the [[Prototype]] of *O* is a host object that does not implement the [[CanPut]] method, return **false**.
6. Call the [[CanPut]] method of [[Prototype]] of *O* with property name *P*.
7. Return Result(6).

8.6.2.4 [[HasProperty]](P)

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

2. If O has a property with name P, return true.
3. If the [[Prototype]] of O is null, return false.
4. Call the [[HasProperty]] method of [[Prototype]] with property name P.
5. Return Result(3).

8.6.2.5 [[Delete]](P)

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

1. If *O* doesn't have a property with name *P*, return **true**.
2. If the property has the DontDelete attribute, return **false**.

3. Remove the property with name *P* from *O*.
4. Return **true**.

8.6.2.6 [[DefaultValue]](hint)

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

1. Call the [[Get]] method of object O with argument "toString".
2. If Result(1) is not an object, go to step 5.
3. Call the [[Call]] method of Result(1), with O as the this value and an empty argument list.
4. If Result(3) is a primitive value, return Result(3).
5. Call the [[Get]] method of object O with argument "valueOf".
6. If Result(5) is not an object, go to step 9.
7. Call the [[Call]] method of Result(5), with O as the this value and an empty argument list.
8. If Result(7) is a primitive value, return Result(7).
9. Generate a runtime error.

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

1. Call the [[Get]] method of object *O* with argument **`"valueOf"`**.
2. If Result(1) is not an object, go to step 5.
3. Call the [[Call]] method of Result(1), with *O* as the **`this`** value and an empty argument list.
4. If Result(3) is a primitive value, return Result(3).
5. Call the [[Get]] method of object *O* with argument **`"toString"`**.
6. If Result(5) is not an object, go to step 9.
7. Call the [[Call]] method of Result(5), with *O* as the **`this`** value and an empty argument list.
8. If Result(7) is a primitive value, return Result(7).
9. Generate a runtime error.

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

8.7 The Reference Type

The internal Reference type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon references in the manner described here. However, a value of type **Reference** is used only as an intermediate result of expression evaluation and cannot be stored as the value of a variable or property.

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.)

Another use of the Reference type is to explain the determination of the **`this`** value for a function call.

A **Reference** is a reference to a property of an object. A Reference consists of two components, the *base object* and the *property name*.

The following abstract operations are used in this specification to describe the behaviour of references:

- GetBase(V). Returns the base object component of the reference V.
- GetPropertyName(V). Returns the property name component of the reference V.
- GetValue(V). Returns the value of the property indicated by the reference V.
- PutValue(V, W). Changes the value of the property indicated by the reference V to be W.

8.7.1 GetBase(V)

1. If Type(V) is Reference, return the base object component of V.
2. Generate a runtime error.

8.7.2 GetPropertyName(V)

1. If Type(V) is Reference, return the property name component of V.
2. Generate a runtime error.

8.7.3 GetValue(V)

1. If Type(V) is not Reference, return V.
2. Call GetBase(V).
3. If Result(2) is **null**, generate a runtime error.
4. Call the [[Get]] method of Result(2), passing GetPropertyName(V) for the property name.
5. Return Result(4).

8.7.4 PutValue(V, W)

1. If Type(V) is not Reference, generate a runtime error.
2. Call GetBase(V).
3. If Result(2) is null, go to step 6.
4. Call the [[Put]] method of Result(2), passing GetPropertyName(V) for the property name and W for the value.
5. Return.
6. Call the [[Put]] method for the global object, passing GetPropertyName(V) for the property name and W for the value.
7. Return.

8.8 The List type

The internal List type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon List values in the manner described here. However, a value of the List type is used only as an intermediate result of expression evaluation and cannot be stored as the value of a variable or property.

The List type is used to explain the evaluation of argument lists (section 11.2.4) in **`new`** expressions and in function calls. Values of the List type are simply ordered sequences of values. These sequences may be of any length.

8.9 The Completion Type

The internal Completion type is not a language data type. It is defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon Completion values in the manner described here. However, a value of the Completion type is used only as an intermediate result of statement evaluation and cannot be stored as the value of a variable or property.

The Completion type is used to explain the behaviour of statements (**`break`**, **`continue`**, and **`return`**) that perform nonlocal transfers of control. Values of the Completion type have one of the following forms:

- "normal completion"
- "normal completion after value *V*"
- "abrupt completion because of **`break`**"
- "abrupt completion after value *V* because of **`break`**"
- "abrupt completion because of **`continue`**"
- "abrupt completion after value *V* because of **`continue`**"
- "abrupt completion because of **`return`** *V*" where *V* is a value

Any completion of one of the four forms that carries a value *V* is called a *value completion*. Any completion of one of the first two forms is called a *normal completion*; any other completion is called an *abrupt completion*. Any completion of a form that mentions **`break`** is called a **`break`** *completion*. Any completion of a form that mentions **`continue`** is called a **`continue`** *completion*. Any completion of a form that mentions **`return`** is called a **`return`** *completion*.

9 Type Conversion

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 operators. These operators are not a part of the language; they are defined here to aid the specification of the semantics of the language. The conversion operators are polymorphic; that is, they can accept a value of any standard type, but not of type Reference, List, or Completion (the internal types).

9.1 ToPrimitive

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

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 internal [[DefaultValue]] method of the object, passing the optional hint *PreferredType*. The behaviour of the [[DefaultValue]] method is defined by this specification for all native ECMAScript objects (see section 8.6.2.6). If the return value is of type Object or Reference, a runtime error is generated.

9.2 ToBoolean

The operator ToBoolean converts its argument to a value of type Boolean according to the following table:

Input 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 operator ToNumber converts its argument to a value of type Number according to the following table:

Input 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. Call ToPrimitive(input argument, hint Number). 2. Call ToNumber(Result(1)). 3. Return Result(2).

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**.

StringNumericLiteral **:::**

StrWhiteSpace$_{opt}$
StrWhiteSpace$_{opt}$ *StrNumericLiteral StrWhiteSpace*$_{opt}$

StrWhiteSpace **:::**

StrWhiteSpaceChar StrWhiteSpace$_{opt}$

StrWhiteSpaceChar **:::**

<TAB>
<SP>
<FF>
<VT>
<CR>
<LF>

StrNumericLiteral **:::**

StrDecimalLiteral
`+` *StrDecimalLiteral*
`-` *StrDecimalLiteral*
HexIntegerLiteral

StrDecimalLiteral **:::**

`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* (section 7.7.3):

- A *StringNumericLiteral* may be preceded and/or followed by whitespace and/or line terminators.
- A StringNumericLiteral may not use octal notation.
- 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 whitespace 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 (section 7.7.3), 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, ideally using IEEE 754 round-to-nearest mode, to a representable value of the number type.

- The MV of *StringNumericLiteral* **:::** (an empty character sequence) 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 whitespace is present or not.
- The MV of *StrNumericLiteral* **:::** *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* **:::** **`+`** *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* **:::** **`-`** *StrDecimalLiteral* is the negative of the MV of *StrDecimalLiteral*. (Note that if the MV of *StrDecimalLiteral* 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 *StrNumericLiteral* **:::** *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StrDecimalLiteral* **:::** **`Infinity`** is 10^{10000} (a value so large that it will round to +∞).
- The MV of *StrDecimalLiteral* **:::** *DecimalDigits* **`.`** is the MV of *DecimalDigits*.
- The MV of *StrDecimalLiteral* **:::** *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 *StrDecimalLiteral* **:::** *DecimalDigits* **.** *ExponentPart* is the MV of *DecimalDigits* times 10^e, where *e* is the MV of *ExponentPart*.
- The MV of *StrDecimalLiteral* **:::** *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 *DecimalDigit*s and *e* is the MV of *ExponentPart*.
- The MV of *StrDecimalLiteral* **::: .** *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n}, where *n* is the number of characters in *DecimalDigit*s.
- The MV of *StrDecimalLiteral* **::: .** *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times 10^{e-n}, where *n* is the number of characters in *DecimalDigit*s and *e* is the MV of *ExponentPart*.
- The MV of *StrDecimalLiteral* **:::** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *StrDecimalLiteral* **:::** *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-whitespace 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 section 8.5), unless the literal includes a StrDecimalLiteral 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 (either 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 operator ToInteger converts its argument to an integral numeric value. This operator functions as follows:

1. Call ToNumber on the input argument.
2. If Result(1) is **NaN**, return **+0**.
3. If Result(1) is **+0**, **−0**, +∞**,** or **−**∞, return Result(1).
4. Compute sign(Result(1)) * floor(abs(Result(1))).
5. Return Result(4).

9.5 ToInt32: (signed 32 bit integer)

The operator ToInt32 converts its argument to one of 2^{32} integer values in the range -2^{31} through $2^{31}-1$, inclusive. This operator functions as follows:

1. Call ToNumber on the input argument.
2. If Result(1) is **NaN**, **+0**, **−0**, **+**∞, or **−**∞, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. Compute Result(3) modulo 2^{32}; that is, a finite integer value k of Number type with positive sign and less than 2^{32} in magnitude such the mathematical difference of Result(3) and k is mathematically an integer multiple of 2^{32}.
5. If Result(4) is greater than or equal to 2^{31}, return Result(4)− 2^{32}, otherwise return Result(4).

NOTE Given the above definition of ToInt32:

The ToInt32 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 operator ToUint32 converts its argument to one of 2^{32} integer values in the range 0 through $2^{32}-1$, inclusive. This operator functions as follows:

1. Call ToNumber on the input argument.
2. If Result(1) is **NaN**, +0, −0, +∞, or −∞, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. Compute Result(3) modulo 2^{32}; that is, a finite integer value k of Number type with positive sign and less than 2^{32} in magnitude such the mathematical difference of Result(3) and k is mathematically an integer multiple of 2^{32}.
5. Return Result(4).

NOTE Given the above definition of ToUInt32::

Step 5 is the only difference between ToUint32 and ToInt32.

The ToUint32 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 operator ToUint16 converts its argument to one of 2^{16} integer values in the range 0 through $2^{16}-1$, inclusive. This operator functions as follows:

1. Call ToNumber on the input argument.
2. If Result(1) is **NaN**, +0, −0, +∞, or −∞, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. Compute Result(3) modulo 2^{16}; that is, a finite integer value k of Number type with positive sign and less than 2^{16} in magnitude such the mathematical difference of Result(3) and k is mathematically an integer multiple of 2^{16}.
5. Return Result(4).

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 ToUnit16.

ToUint16 maps −0 to +0.

9.8 ToString

The operator ToString converts its argument to a value of type String according to the following table:

Input 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 note below.
String	Return the input argument (no conversion)
Object	Apply the following steps: 1. Call ToPrimitive(input argument, hint String). 2. Call ToString(Result(1)). 3. Return Result(2).

9.8.1 ToString Applied to the Number Type

The operator 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 $k = 1$, $10^{k-1} = s < 10^k$, the number value for $s \cdot 10^{n-k}$ 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 $k = n = 21$, 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 = 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 = 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 zeros).
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 zeros).

NOTE 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.

For implementations which 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 be integers such that $k \geq 1$, $10^{k-1} = s < 10^k$, the number value for $s \cdot 10^{n-k}$ is *m*, and *k* is as small as possible. If there are multiple possibilities for *s*, choose the value of *s* for which $s \cdot 10^{n-k}$ is closest in value to *m*. Note that *k* is the number of digits in the decimal representation of *s*, that *s* is not divisible by 10. If there are two such possible values of *s*, choose the one that is even.

Implementors 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 operator ToObject converts its argument to a value of type Object according to the following table:

Input Type	*Result*
Undefined	Generate a runtime error.
Null	Generate a runtime error.
Boolean	Create a new boolean object whose default value is the value of the boolean. See section 15.6 for a description of boolean objects.
Number	Create a new number object whose default value is the value of the number. See section 15.7 for a description of number objects.
String	Create a new string object whose default value is the value of the string. See section 15.5 for a description of string objects.
Object	The result is the input argument (no conversion).

10 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.

10.1 Definitions

10.1.1 Function Objects

There are four types of function objects:

- Declared functions are defined in source text by a *FunctionDeclaration*.
- Anonymous functions are created dynamically by using the built-in **`Function`** object as a constructor, which is referred to as an instantiating **`Function`**.
- Implementation-supplied functions are created at the request of the host with source text supplied by the host. The mechanism for their creation is implementation-dependent. Implementation-supplied functions may have any subset of the following attributes {ImplicitThis, ImplicitParents }. Note that these are attributes of function objects, not of properties. The use of these attributes is described in section 10.2.4.
- Internal functions are built-in objects of the language, such as **`parseInt`** and **`Math.exp`**. An implementation may also provide implementation-dependent internal functions that are not described in this specification. These functions do not contain executable code defined by the ECMAScript grammar, so are excluded from this discussion of execution contexts.

10.1.2 Types of Executable Code

There are five types of executable ECMAScript source text:

- *Global code* is source text that is outside all function declarations. More precisely, the global code of a particular ECMAScript *Program* consists of all *SourceElements* in the *Program* production, which come from the *Statement* definition.
- *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 the string parameter.
- *Function code* is source text that is inside a function declaration. More precisely, the function code of a particular ECMAScript *FunctionDeclaration* consists of the *Block* in the definition of *FunctionDeclaration*.
- *Anonymous code* is the source text supplied when instantiating **`Function`**. More precisely, the last parameter provided in an instantiation of **`Function`** is converted to a string and treated as the *StatementList* of the *Block* of a *FunctionDeclaration*. If more than one parameter is provided in an instantiation of **`Function`**, all parameters except the last one are converted to strings and concatenated together, separated by commas. The resulting string is interpreted as the *FormalParameterList* of a *FunctionDeclaration* for the *StatementList* defined by the last parameter.
- *Implementation-supplied code* is the source text supplied by the host when creating an implementation-supplied function. The source text is treated as the *StatementList* of the *Block* of a *FunctionDeclaration*. Depending on the implementation, the host may also supply a *FormalParameterList*.

10.1.3 Variable instantiation

Every execution context has associated with it a variable object. Variables declared in the source text are added as properties of the variable object. For global and eval code, functions defined in the source text are added as properties of the variable object. Function declarations in other types of code are not allowed by the grammar. For function, anonymous, and implementation-supplied code, parameters are added as properties of the variable object.

Which object is used as the variable object and what attributes are used for the properties depends on the type of code, but the remainder of the behaviour is generic:

- For each *FunctionDeclaration* in the code, in source text order, instantiate a declared function from the *FunctionDeclaration* and create a property of the variable object whose name is the Identifier in the *FunctionDeclaration*, whose value is the declared function and whose attributes are determined by the type of code. If the variable object already has a property with this name, replace its value and attributes.
- For each formal parameter, as defined in the *FormalParameterList*, create a property of the variable object whose name is the *Identifier* and whose attributes are determined by the type of code. The values of the parameters are supplied by the caller. If the caller supplies fewer parameter values than there are formal parameters, the extra formal parameters have value **`undefined`**. If two or more formal parameters share the same name, hence the same property, the corresponding property is given the value that was supplied for the last parameter with this name. If the value of this last parameter was not supplied by the caller, the value of the corresponding property is **`undefined`**.
- For each *VariableDeclaration* in the code, create a property of the variable object whose name is the *Identifier* in *VariableDeclaration*, whose value is **`undefined`** and whose attributes are determined by the type of code. If there is already a property of the variable object with the name of a declared variable, the value of the property and its attributes are not changed. Semantically, this step must follow the creation of the *FunctionDeclaration* and *FormalParameterList* properties. In particular, if a declared variable has the same name as a declared function or formal parameter, the variable declaration does not disturb the existing property.

10.1.4 Scope Chain and Identifier Resolution

Every execution context has associated with it a *scope chain*. This is logically a list of objects that are searched when *binding* an *Identifier*. When control enters an execution context, the scope chain is created and is populated with an initial set of objects, depending on the type of code. When control leaves the execution context, the scope chain is destroyed.

During execution, the scope chain of the execution context is affected only by *WithStatement*. When execution enters a **`with`** block, the object specified in the **`with`** statement is added to the front of the scope chain. When execution leaves a **`with`** block, whether normally or via a **`break`** or **`continue`** statement, the object is removed from the scope chain. The object being removed will always be the first object in the scope chain.

During execution, the syntactic production *PrimaryExpression* : *Identifier* is evaluated using the following algorithm:

1. Get the next object in the scope chain. If there isn't one, go to step 5.
2. Call the [[HasProperty]] method of Result(l), passing the *Identifier* as the property.
3. If Result(2) is **`true`**, return a value of type Reference whose base object is Result(l) and whose property name is the *Identifier*.
4. Go to step 1.
5. Return a value of type Reference whose base object is **`null`** and whose property name is the *Identifier*.

The result of binding an identifier is always a value of type Reference with its member name component equal to the identifier string.

10.1.5 Global Object

There is a unique *global object*, which is created before control enters any execution context. Initially the global object has the following properties:

- Built-in objects such as Math, String, Date, parseInt, etc. These have attributes { DontEnum }.
- Additional host defined properties. This may include a property whose value is the global object itself, for example **`window`** in HTML.

As control enters execution contexts, and as ECMAScript code is executed, additional properties may be added to the global object and the initial properties may be changed.

10.1.6 Activation object

When control enters an execution context for declared function code, anonymous code or implementation-supplied code, an object called the activation object is created and associated with the execution context. The activation object is initialised with a property with name **`arguments`** and property attributes { DontDelete }. The initial value of this property is the arguments object described below.

The activation object is then used as the variable object for the purposes of variable instantiation.

When a value is to be returned from the call to a function, its activation object is no longer needed and may be permanently decommissioned.

The activation object is purely a specification mechanism. It is impossible for an ECMAScript program to access the activation object. It can access members of the activation object, but not the activation object itself. When the call operation is applied to a Reference value whose base object is an activation object, **`null`** is used as the **`this`** value of the call.

10.1.7 This

There is a **`this`** value associated with every active execution context. The **`this`** value depends on the caller and the type of code being executed and is determined when control enters the execution context. The **`this`** value associated with an execution context is immutable.

10.1.8 Arguments Object

When control enters an execution context for declared function code, anonymous code, or implementation-supplied code, an arguments object is created and initialised as follows:

- The value of the internal [[Prototype]] property of the arguments object is the original Object prototype object, the one that is the initial value of **`Object.prototype`** (section 15.2.3.1).
- A property is created with name **`callee`** and property attributes { DontEnum }. The initial value of this property is the function object being executed. This allows anonymous functions to be recursive.
- A property is created with name **`length`** and property attributes { DontEnum }. The initial value of this property is the number of actual parameter values supplied by the caller.
- For each non-negative integer, *iarg*, less than the value of the **`length`** property, a property is created with name ToString(*iarg*) and property attributes { DontEnum }. The initial value of this property is the value of the corresponding actual parameter supplied by the caller. The first actual parameter value corresponds to *iarg* = 0, the second to *iarg* = 1 and so on. In the case when *iarg* is less than the number of formal parameters for the function object, this property shares its value with the corresponding property of the activation object. This means that changing this property changes the corresponding property of the activation object and vice versa. The value sharing mechanism depends on the implementation.

10.2 Entering An Execution Context

When control enters an execution context, the scope chain is created and initialised, variable instantiation is performed, and the **`this`** value is determined.

The initialisation of the scope chain, variable instantiation, and the determination of the **`this`** value depend on the type of code being entered.

10.2.1 Global Code

- The scope chain is created and initialised to contain the global object and no others.
- Variable instantiation is performed using the global object as the variable object and using empty property attributes.
- The **`this`** value is the global object.

10.2.2 Eval Code

When control enters an execution context for eval code, the previous active execution context, referred to as the *calling context*, is used to determine the scope chain, the variable object, and the **`this`** value.

If there is no calling context, then initialising the scope chain, variable instantiation, and determination of the `this` value are performed just as for global code.

- The scope chain is initialised to contain the same objects, in the same order, as the calling context's scope chain. This includes objects added to the calling context's scope chain by *WithStatement.*
- Variable instantiation is performed using the calling context's variable object and using empty property attributes.
- The `this` value is the same as the `this` value of the calling context.

10.2.3 Function and Anonymous Code

- The scope chain is initialised to contain the activation object followed by the global object.
- Variable instantiation is performed using the activation object as the variable object and using property attributes { DontDelete }.
- The caller provides the `this` value. If the `this` value provided by the caller is not an object (including the case where it is `null`), then the `this` value is the global object.

10.2.4 Implementation-supplied Code

- The scope chain is initialised to contain the activation object as its first element.
- The `this` value is determined just as for function and anonymous code.
- If the implementation-supplied function has the ImplicitThis attribute (10.1.1), the `this` value is placed in the scope chain after the activation object.
- If the implementation-supplied function has the ImplicitParents attribute (10.1.1), a list of objects, determined solely by the `this` value, is inserted in the scope chain after the activation object (if the implementation-supplied function does not have the ImplicitThis attribute) or after the activation object and `this` object (if the implementation-supplied function has the ImplicitThis attribute). Note that this list is determined at run time by the `this` value. It is not determined by any form of lexical scoping.
- The global object is placed in the scope chain after all other objects.
- Variable instantiation is performed using the activation object as the variable object and using attributes { DontDelete }.

11 Expressions

11.1 Primary Expressions

Syntax

PrimaryExpression **:**

this
Identifier
Literal
(*Expression* **)**

11.1.1 The `this` keyword

The `this` keyword evaluates to the `this` value of the execution context.

11.1.2 Identifier reference

An *Identifier* is evaluated using the scoping rules stated in section 10.1.4. The result of an *Identifier* is always a value of type Reference.

11.1.3 Literal reference

A *Literal* is evaluated as described in section 7.7.

11.1.4 The Grouping Operator

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

1. Evaluate Expression. This may be of type Reference.
2. Return Result(1).

NOTE This algorithm does not apply GetValue to Result(1). 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
MemberExpression **[** *Expression* **]**
MemberExpression **.** *Identifier*
new *MemberExpression Arguments*

NewExpression **:**

MemberExpression
new *NewExpression*

CallExpression **:**

MemberExpression Arguments
CallExpression Arguments
CallExpression **[** *Expression* **]**
CallExpression **.** *Identifier*

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 **.** *Identifier*
CallExpression **.** *Identifier*

or the bracket notation:

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

The dot notation is explained by the following syntactic conversion:

MemberExpression ***.*** Identifier

is identical in its behaviour to

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

and similarly

CallExpression ***.*** Identifier

is identical in its behaviour to

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

where <identifier-string> is a string literal containing the same sequence of characters as the *Identifier*.

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

1. Evaluate MemberExpression.
2. Call GetValue(Result(1)).
3. Evaluate Expression.
4. Call GetValue(Result(3)).
5. Call ToObject(Result(2)).
6. Call ToString(Result(4)).
7. Return a value of type Reference whose base object is Result(5) and whose property name is Result(6).

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. Evaluate NewExpression.
2. Call GetValue(Result(1)).
3. If Type(Result(2)) is not Object, generate a runtime error.
4. If Result(2) does not implement the internal [[Construct]] method, generate a runtime error.
5. Call the [[Construct]] method on Result(2), providing no arguments (that is, an empty list of arguments).
6. If Type(Result(5)) is not Object, generate a runtime error.
7. Return Result(5).

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

1. Evaluate MemberExpression.
2. Call GetValue(Result(1)).
3. Evaluate Arguments, producing an internal list of argument values (section 11.2.4).
4. If Type(Result(2)) is not Object, generate a runtime error.
5. If Result(2) does not implement the internal [[Construct]] method, generate a runtime error.
6. Call the [[Construct]] method on Result(2), providing the list Result(3) as the argument values.
7. If Type(Result(6)) is not Object, generate a runtime error.
8. Return Result(6).

11.2.3 Function Calls

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

1. Evaluate MemberExpression.
2. Evaluate Arguments, producing an internal list of argument values (section 11.2.4).
3. Call GetValue(Result(1)).
4. If Type(Result(3)) is not Object, generate a runtime error.
5. If Result(3) does not implement the internal [[Call]] method, generate a runtime error.
6. If Type(Result(1)) is Reference, Result(6) is GetBase(Result(1)). Otherwise, Result(6) is **null**.
7. If Result(6) is an activation object, Result(7) is **null**. Otherwise, Result(7) is the same as Result(6).
8. Call the [[Call]] method on Result(3), providing Result(7) as the **`this`** value and providing the list Result(2) as the argument values.
9. Return Result(8).

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

NOTE Result(8) will never be of type Reference if Result(3) is a native ECMAScript object. Whether calling a host object can return a value of type Reference is implementation-dependent.

11.2.4 Argument Lists

The evaluation of an argument list produces an internal list of values (section 8).

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

1. Return an empty internal list of values.

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

1. Evaluate ArgumentList.
2. Return Result(1).

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

1. Evaluate AssignmentExpression.
2. Call GetValue(Result(1)).
3. Return an internal list whose sole item is Result(2).

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

1. Evaluate ArgumentList.
2. Evaluate AssignmentExpression.
3. Call GetValue(Result(2)).
4. Return an internal list whose length is one greater than the length of Result(1) and whose items are the items of Result(1), in order, followed at the end by Result(3), which is the last item of the new list.

11.3 Postfix expressions

Syntax

PostfixExpression **:**

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

11.3.1 Postfix increment operator

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

1. Evaluate MemberExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Add the value **1** to Result(3), using the same rules as for the **+** operator (section Applying the additive operators (**+, -**) to numbers).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(3).

11.3.2 Postfix decrement operator

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

1. Evaluate MemberExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Subtract the value **1** from Result(3), using the same rules as for the **-** operator (section Applying the additive operators (**+, -**) to numbers).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(3).

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. Evaluate UnaryExpression.
2. Call GetBase(Result(1)).
3. Call GetPropertyName(Result(1)).
4. If Type(Result(2)) is not Object, return **true**.
5. If Result(2) does not implement the internal [[Delete]] method, go to step 8.
6. Call the [[Delete]] method on Result(2), providing Result(3) as the property name to delete.
7. Return Result(6).
8. Call the [[HasProperty]] method on Result(2), providing Result(3) as the property name to check for.
9. If Result(8) is **true**, return **false**.
10. Return **true**.

11.4.2 The `void` operator

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

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Return **undefined**.

11.4.3 The `typeof` operator

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

1. Evaluate UnaryExpression.
2. If Type(Result(1)) is Reference and GetBase(Result(1)) is **null**, return **`"undefined"`**.
3. Call GetValue(Result(1)).
4. Return a string determined by Type(Result(3)) according to the following table:

Type	***Result***
Undefined	**`"undefined"`**
Null	**`"object"`**
Boolean	**`"boolean"`**
Number	**`"number"`**
String	**`"string"`**
Object (native and doesn’t implement [[Call]])	**`"object"`**
Object (native and implements [[Call]])	**`"function"`**
Object (host)	Implementation-dependent

11.4.4 Prefix increment operator

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

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Add the value **`1`** to Result(3), using the same rules as for the **+** operator (section 11.6.3).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(4).

11.4.5 Prefix decrement operator

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

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Subtract the value **`1`** from Result(3), using the same rules as for the **-** operator (section 11.6.3).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(4).

11.4.6 Unary + operator

The unary + operator converts its operand to Number type.

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

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Return Result(3).

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. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. If Result(3) is **NaN**, return **NaN**.
5. Negate Result(3); that is, compute a number with the same magnitude but opposite sign.
6. Return Result(5).

11.4.8 The bitwise NOT operator (~)

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

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToInt32(Result(2)).
4. Apply bitwise complement to Result(3). The result is a signed 32-bit integer.
5. Return Result(4).

11.4.9 Logical NOT operator (!)

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

1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **true**, return **false**.
5. 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. Evaluate MultiplicativeExpression.
2. Call GetValue(Result(1)).
3. Evaluate UnaryExpression.
4. Call GetValue(Result(3)).
5. Call ToNumber(Result(2)).
6. Call ToNumber(Result(4)).
7. Apply the specified operation (*, /, or %) to Result(5) and Result(6). See the notes below (11.5.1, 11.5.2, 11.5.3).
8. Return Result(7).

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 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 non-zero 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 non-zero 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 non-zero 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 binary **%** operator is said to yield the remainder of its operands from an implied division; the left operand is the dividend and the right operand is the divisor. In C and C++, the remainder operator accepts only integral operands, but 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 a 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 finite, the result is the same as the dividend.

- In the remaining cases, where neither an infinity, nor a zero, nor **NaN** is involved, the floating-point remainder r from a dividend n and a divisor d is defined by the mathematical relation r = n − (d * q) where q is an integer that is negative only if n/d is negative and positive only if n/d is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of n and d.

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. Evaluate AdditiveExpression.
2. Call GetValue(Result(1)).
3. Evaluate MultiplicativeExpression.
4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2)).
6. Call ToPrimitive(Result(4)).
7. If Type(Result(5)) is String or Type(Result(6)) is String, go to step 12. (Note that this step differs from step 3 in the algorithm for comparison for the relational operators in using or instead of and.)
8. Call ToNumber(Result(5)).
9. Call ToNumber(Result(6)).
10. Apply the addition operation to Result(8) and Result(9). See the note below (11.6.3).
11. Return Result(10).
12. Call ToString(Result(5)).
13. Call ToString(Result(6)).
14. Concatenate Result(12) followed by Result(13).
15. Return Result(14).
```

NOTE 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.

11.6.2 The subtraction operator (-)

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

```
1. Evaluate AdditiveExpression.
2. Call GetValue(Result(1)).
3. Evaluate MultiplicativeExpression.
4. Call GetValue(Result(3)).
5. Call ToNumber(Result(2)).
6. Call ToNumber(Result(4)).
7. Apply the subtraction operation to Result(5) and Result(6). See the note below (11.6.3).
8. Return Result(7).
```

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 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 zeros is **–0**. The sum of two positive zeros, or of two zeros 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*

Semantics

The result of evaluating *ShiftExpression* is always truncated to 32 bits. If the result of evaluating *ShiftExpression* produces a fractional component, the fractional component is discarded. The result of evaluating an *AdditiveExpresion* that is the right-hand operand of a shift operator is always truncated to five bits.

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. Evaluate ShiftExpression.
2. Call GetValue(Result(1)).
3. Evaluate AdditiveExpression.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToUint32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Left shift Result(5) by Result(7) bits. The result is a signed 32 bit integer.
9. Return Result(8).

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. Evaluate ShiftExpression.
2. Call GetValue(Result(1)).
3. Evaluate AdditiveExpression.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToUint32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Perform sign-extending right shift of Result(5) by Result(7) bits. The most significant bit is propagated. The result is a signed 32 bit integer.
9. Return Result(8).

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. Evaluate ShiftExpression.
2. Call GetValue(Result(1)).
3. Evaluate AdditiveExpression.
4. Call GetValue(Result(3)).
5. Call ToUint32(Result(2)).
6. Call ToUint32(Result(4)).
7. Mask out all but the least significant 5 bits of Result(6), that is, compute Result(6) & 0x1F.
8. Perform zero-filling right shift of Result(5) by Result(7) bits. Vacated bits are filled with zero. The result is an unsigned 32 bit integer.
9. Return Result(8).

11.8 Relational operators

Syntax

RelationalExpression **:**

ShiftExpression
RelationalExpression **<** *ShiftExpression*
RelationalExpression **>** *ShiftExpression*
RelationalExpression **<=** *ShiftExpression*
RelationalExpression **>=** *ShiftExpression*

Semantics

The result of evaluating *RelationalExpression* is always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.

11.8.1 The less-than operator (<)

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

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(2) < Result(4). (See section 11.8.5).
6. If Result(5) is **undefined**, return **false**. Otherwise, return Result(5).

11.8.2 The greater-than operator (>)

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

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).

5. Perform the comparison Result(4) < Result(2). (See section 11.8.5).
6. If Result(5) is **undefined**, return **false**. Otherwise, return Result(5).

11.8.3 The less-than-or-equal operator (<=)

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

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) < Result(2). (See section 11.8.5).
6. If Result(5) 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. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(2) < Result(4). (See section 11.8.5).
6. If Result(5) 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**). Such a comparison is performed as follows:

1. Call ToPrimitive(*x*, hint Number).
2. Call ToPrimitive(*y*, hint Number).
3. If Type(Result(1)) is String and Type(Result(2)) is String, go to step 16. (Note that this step differs from step 7 in the algorithm for the addition operator **+** in using *and* instead of *or*.)
4. Call ToNumber(Result(1)).
5. Call ToNumber(Result(2)).
6. If Result(4) is **NaN**, return **undefined**.
7. If Result(5) is **NaN**, return **undefined**.
8. If Result(4) and Result(5) are the same number value, return **false**.
9. If Result(4) is **+0** and Result(5) is **−0**, return **false**.
10. If Result(4) is **−0** and Result(5) is **+0**, return **false**.
11. If Result(4) is +∞, return **false**.
12. If Result(5) is +∞, return **true**.
13. If Result(5) is −∞, return **false**.
14. If Result(4) is −∞, return **true**.
15. If the mathematical value of Result(4) is less than the mathematical value of Result(5)—note that these mathematical values are both finite and not both zero—return **true**. Otherwise, return **false**.
16. If Result(2) is a prefix of Result(1), 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.)
17. If Result(1) is a prefix of Result(2), return **true**.
18. Let *k* be the smallest nonnegative integer such that the character at position *k* within Result(1) is different from the character at position *k* within Result(2). (There must be such a *k*, for neither string is a prefix of the other.)
19. Let *m* be the integer that is the Unicode encoding for the character at position *k* within Result(1).
20. Let *n* be the integer that is the Unicode encoding for the character at position *k* within Result(2).
21. If $m < n$, return **true**. Otherwise, return **false**.

NOTE The comparison of strings uses a simple lexicographic ordering on sequences of Unicode code point 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 2.0 specification.

11.9 Equality operators

Syntax

EqualityExpression **:**

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

Semantics

The result of evaluating *EqualityExpression* is always of type Boolean, reflecting whether the relationship named by the operator holds between its two operands.

11.9.1 The equals operator (==)

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

1. Evaluate EqualityExpression.
2. Call GetValue(Result(1)).
3. Evaluate RelationalExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) == Result(2). (See section 11.9.3).
6. Return Result(5).

11.9.2 The does-not-equals operator (!=)

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

1. Evaluate EqualityExpression.
2. Call GetValue(Result(1)).
3. Evaluate RelationalExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) == Result(2). (See section 11.9.3).
6. If Result(5) 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 different from Type(*y*), go to step 14.
2. If Type(*x*) is Undefined, return **true**.
3. If Type(*x*) is Null, return **true**.
4. If Type(*x*) is not Number, go to step 11.
5. If *x* is **NaN**, return **false**.
6. If *y* is **NaN**, return **false**.
7. If *x* is the same number value as *y*, return **true**.
8. If *x* is **+0** and *y* is **−0**, return **true**.
9. If *x* is **−0** and *y* is **+0**, return **true**.
10. Return **false**.
11. 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**..
12. If Type(*x*) is Boolean, return **true** if *x* and *y* are both **true** or both **false**. Otherwise, return **false**.
13. Return **true** if *x* and *y* refer to the same object. Otherwise, return **false**.
14. If *x* is **null** and *y* is **undefined**, return **true**.
15. If *x* is **undefined** and *y* is **null**, return **true**.

16. If Type(*x*) is Number and Type(*y*) is String,
    return the result of the comparison *x* == ToNumber(*y*).
17. If Type(*x*) is String and Type(*y*) is Number,
    return the result of the comparison ToNumber(*x*) == *y*.
18. If Type(*x*) is Boolean, return the result of the comparison ToNumber(*x*) == *y*.
19. If Type(*y*) is Boolean, return the result of the comparison *x* == ToNumber(*y*).
20. If Type(*x*) is either String or Number and Type(*y*) is Object,
    return the result of the comparison *x* == ToPrimitive(*y*).
21. If Type(*x*) is Object and Type(*y*) is either String or Number,
    return the result of the comparison ToPrimitive(*x*) == *y*.
22. Return **false**.

NOTE Given the above definition of equality::

String comparison can be forced by: **`"" + a == "" + b`**.

Numeric comparison can be forced by: **`a - 0 == b - 0`**.

Boolean comparison can be forced by: **`!a == !b`**.

The equality operators maintain the following invariants:

1. **`A != B`** is equivalent to **`!(A == B)`**.
2. **`A == B`** is equivalent to **`B == A`**, except in the order of evaluation of **`A`** and **`B`**.

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.

Comparison of strings uses a simple equality test on sequences of Unicode code point 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 2.0 specification.

11.10 Binary bitwise operators

Syntax

BitwiseANDExpression **:**

EqualityExpression
BitwiseANDExpression **&** *EqualityExpression*

BitwiseXORExpression **:**

BitwiseANDExpression
BitwiseXORExpression **^** *BitwiseANDExpression*

BitwiseORExpression **:**

BitwiseXORExpression
BitwiseORExpression **|** *BitwiseXORExpression*

Semantics

The production *A* **:** *A* @ *B*, where @ is one of the bitwise operators in the productions above, is evaluated as follows:

1. Evaluate *A*.
2. Call GetValue(Result(1)).
3. Evaluate *B*.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToInt32(Result(4)).
7. Apply the bitwise operator @ to Result(5) and Result(6). The result is a signed 32 bit integer.

8. Return Result(7).

11.11 Binary logical operators

Syntax

LogicalANDExpression **:**

BitwiseORExpression
LogicalANDExpression **&&** *BitwiseORExpression*

LogicalORExpression **:**

LogicalANDExpression
LogicalORExpression **||** *LogicalANDExpression*

Semantics

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

1. Evaluate LogicalANDExpression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, return Result(2).
5. Evaluate BitwiseORExpression.
6. Call GetValue(Result(5)).
7. Return Result(6).

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

1. Evaluate LogicalORExpression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is true, return Result(2).
5. Evaluate LogicalANDExpression.
6. Call GetValue(Result(5)).
7. Return Result(6).

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*

Semantics

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

1. Evaluate LogicalORExpression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, go to step 8.
5. Evaluate the first AssignmentExpression.
6. Call GetValue(Result(5)).
7. Return Result(6).

8. Evaluate the second AssignmentExpression.
9. Call GetValue(Result(8)).
10. Return Result(9).

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 AssignmentOperator AssignmentExpression

AssignmentOperator **:: one of**

= ***=** **/=** **%=** **+=** **-=** **<<=** **>>=** **>>>=** **&=** **^=** **|=**

11.13.1 Simple Assignment (=)

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

1. Evaluate LeftHandSideExpression.
2. Evaluate AssignmentExpression.
3. Call GetValue(Result(2)).
4. Call PutValue(Result(1), Result(3)).
5. Return Result(3).

11.13.2 Compound assignment (op=)

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

1. Evaluate LeftHandSideExpression.
2. Call GetValue(Result(1)).
3. Evaluate AssignmentExpression.
4. Call GetValue(Result(3)).
5. Apply operator @ to Result(2) and Result(4).
6. Call PutValue(Result(1), Result(5)).
7. Return Result(5).

11.14 Comma operator (,)

Syntax

Expression **:**

AssignmentExpression
Expression **,** *AssignmentExpression*

Semantics

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

1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Evaluate AssignmentExpression.
4. Call GetValue(Result(3)).
5. Return Result(4).

12 Statements

Syntax

Statement **:**

Block
VariableStatement
EmptyStatement
ExpressionStatement
IfStatement
IterationStatement
ContinueStatement
BreakStatement
ReturnStatement
WithStatement

12.1 Block

Syntax

Block **:**

{ *StatementList*$_{opt}$ **}**

StatementList **:**

Statement
StatementList Statement

Semantics

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

1. Return “normal completion”.

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

1. Evaluate StatementList.
2. Return Result(1).

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

1. Evaluate *Statement*.
2. Return Result(1).

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

1. Evaluate StatementList.
2. If Result(1) is an abrupt completion, return Result(1).
3. Evaluate Statement.
4. If Result(3) is a value completion, return Result(3).
5. If Result(1) is not a value completion, return Result(3).
6. Let V be the value carried by Result(1).
7. If Result(3) is “abrupt completion because of **`break`**”,
   return “abrupt completion after value V because of **`break`**”.
8. If Result(3) is “abrupt completion because of **`continue`**”,
   return “abrupt completion after value V because of **`continue`**”.
9. Return “normal completion after value V”.

12.2 Variable statement

Syntax

VariableStatement **:**

  `var` *VariableDeclarationList* **`;`**

VariableDeclarationList **:**

  VariableDeclaration
  VariableDeclarationList **`,`** *VariableDeclaration*

VariableDeclaration **:**

  Identifier Initializer$_{opt}$

Initializer **:**

  `=` *AssignmentExpression*

Description

If the variable statement occurs inside a *FunctionDeclaration*, the variables are defined with function-local scope in that function, as described in section 10.1.3. Otherwise, they are defined with global scope (that is, they are created as members of the global object, as described in section 10.1.3) using property attributes { DontDelete }.. Variables are created when the execution scope is entered. A *Block* does not define a new execution scope. Only *Program* and *FunctionDeclaration* produce a new scope. Variables are initialised to the **`undefined`** value when created. A variable with an *Initializer* 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 completion”.

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 Initializer* is evaluated as follows:

1. Evaluate *Identifier* as described in section 11.1.2.
2. Evaluate Initializer.
3. Call GetValue(Result(2)).
4. Call PutValue(Result(1), Result(3)).
5. Return a string value containing the same sequence of characters as in the *Identifier*.

The production *Initializer* **:** **`=`** *AssignmentExpression* is evaluated as follows:

1. Evaluate AssignmentExpression.
2. Return Result(1).

12.3 Empty statement

Syntax

EmptyStatement **:**

    `;`

Semantics

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

1. Return “normal completion”.

12.4 Expression statement

Syntax

ExpressionStatement **:**

    Expression `;`

Semantics

The production *ExpressionStatement* **:** *Expression* `;` is evaluated as follows:

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Return “normal completion after value V”, where the value V is Result(2).

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. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **false**, go to step 7.
5. Evaluate the first *Statement*.
6. Return Result(5).
7. Evaluate the second *Statement*.
8. Return Result(7).

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

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is **false**, return “normal completion”.
5. Evaluate *Statement*.
6. Return Result(5).

12.6 Iteration statements

An iteration statement consists of a *header* (which consists of a keyword and a parenthesized control construct) and a *body* (which consists of a *Statement*).

Syntax

IterationStatement **:**

> **`while (`** *Expression* **`)`** *Statement*
> **`for (`** *Expression*$_{opt}$ **`;`** *Expression*$_{opt}$ **`;`** *Expression*$_{opt}$ **`)`** *Statement*
> **`for ( var`** *VariableDeclarationList* **`;`** *Expression*$_{opt}$ **`;`** *Expression*$_{opt}$ **`)`** *Statement*
> **`for (`** *LeftHandSideExpression* **`in`** *Expression* **`)`** *Statement*
> **`for ( var`** *Identifier Initializer*$_{opt}$ **`in`** *Expression* **`)`** *Statement*

12.6.1 The `while` statement

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

1. Let *C* be “normal completion”.
2. Evaluate *Expressio*n.
3. Call GetValue(Result(1)).
4. Call ToBoolean(Result(2)).
5. If Result(3) is **false**, go to step 12.
6. Evaluate *Statement*.
7. If Result(6) is a value completion, change *C* to be “normal completion after value *V*” where *V* is the value carried by Result(6).
8. If Result(6) is a **`break`** completion, go to step 12.
9. If Result(6) is a **`continue`** completion, go to step 2.
10. If Result(6) is a **`return`** completion, return Result(6).
11. Go to step 2.
12. Return *C*.

12.6.2 The `for` statement

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

1. If the first *Expression* is not present, go to step 4.
2. Evaluate the first *Expression*.
3. Call GetValue(Result(2)). (This value is not used.)
4. Let *C* be “normal completion”.
5. If the second *Expression* is not present, go to step 10.
6. Evaluate the second *Expressio*n.
7. Call GetValue(Result(6)).
8. Call ToBoolean(Result(7)).
9. If Result(8) is **false**, go to step 19.
10. Evaluate *Statement*.
11. If Result(10) is a value completion, change *C* to be “normal completion after value *V*” where *V* is the value carried by Result(10).
12. If Result(10) is a **`break`** completion, go to step 19.
13. If Result(10) is a **`continue`** completion, go to step 15.
14. If Result(10) is a **`return`** completion, return Result(10).
15. If the third *Expression* is not present, go to step 5.
16. Evaluate the third *Expressio*n.
17. Call GetValue(Result(16). (This value is not used.)
18. Go to step 5.
19. Return *C*.

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

1. Evaluate VariableDeclarationList.
2. Let C be “normal completion”.
3. If the second Expression is not present, go to step 8.
4. Evaluate the second Expression.

5. Call GetValue(Result(4)).
6. Call ToBoolean(Result(5)).
7. If Result(6) is **false**, go to step 15.
8. Evaluate Statement.
9. If Result(8) is a value completion, change C to be “normal completion after value V” where V is the value carried by Result(8).
10. If Result(8) is a **`break`** completion, go to step 17.
11. If Result(8) is a **`continue`** completion, go to step 13.
12. If Result(8) is a **`return`** completion, return Result(8).
13. If the third Expression is not present, go to step 3.
14. Evaluate the third Expression.
15. Call GetValue(Result(14)). (This value is not used.)
16. Go to step 3.
17. Return *C*.

12.6.3 The `for..in` statement

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

1. Evaluate the *Expression*.
2. Call GetValue(Result(1)).
3. Call ToObject(Result(2)).
4. Let *C* be “normal completion”.
5. Get the name of the next property of Result(3) that doesn’t have the DontEnum attribute. If there is no such property, go to step 14.
6. Evaluate the *LeftHandSideExpression* (it may be evaluated repeatedly).
7. Call PutValue(Result(6), Result(5)).
8. Evaluate *Statement*.
9. If Result(8) is a value completion, change C to be “normal completion after value V” where V is the value carried by Result(8).
10. If Result(8) is a **`break`** completion, go to step 14.
11. If Result(8) is a **`continue`** completion, go to step 5.
12. If Result(8) is a **`return`** completion, return Result(8).
13. Go to step 5.
14. Return *C*.

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

1. Evaluate VariableDeclaration.
2. Evaluate Expression.
3. Call GetValue(Result(2)).
4. Call ToObject(Result(3)).
5. Let *C* be “normal completion”.
6. Get the name of the next property of Result(4) that doesn’t have the DontEnum attribute. If there is no such property, go to step 15.
7. Evaluate Result(1) as if it were an Identifier; see 11.1.2 (yes, it may be evaluated repeatedly).
8. Call PutValue(Result(7), Result(6)).
9. Evaluate Statement.
10. If Result(9) is a value completion, change C to be “normal completion after value V” where V is the value carried by Result(9).
11. If Result(9) a **`break`** completion, go to step 15.
12. If Result(9) a **`continue`** completion, go to step 6.
13. If Result(9) a **`return`** completion, return Result(9).
14. Go to step 6.
15. Return *C*.

The mechanics of enumerating the properties (step 5 in the first algorithm, step 10 in the second) is implementation-dependent. The order of enumeration is defined by the object. 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.

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.

12.7 The `continue` statement

Syntax

ContinueStatement **:**

`continue ;`

Semantics

An ECMAScript program is considered syntactically incorrect if it contains a **`continue`** statement that is not within a **`while`** or **`for`** statement. The **`continue`** statement is evaluated as:

1. Return "abrupt completion because of **`continue`**".

12.8 The `break` statement

Syntax

BreakStatement **:**

`break ;`

Semantics

An ECMAScript program is considered syntactically incorrect if it contains a **`break`** statement that is not within a **`while`** or **`for`** statement. The **`break`** statement is evaluated as:

1. Return "abrupt completion because of **`break`**".

12.9 The `return` statement

Syntax

ReturnStatement **:**

`return` [no *LineTerminator* here] *Expression*$_{opt}$ **`;`**

Semantics

An ECMAScript program is considered syntactically incorrect if it contains a **`return`** statement that is not within the *Block* of a *FunctionDeclaration*. It causes a function to cease execution and return a value to the caller. If *Expression* is omitted, the return value is the **`undefined`** value. Otherwise, the return value is the value of *Expression*.

The production *ReturnStatement* **::** **`return`** [no *LineTerminator* here] *Expression*$_{opt}$ **`;`** is evaluated as:

1. If the *Expression* is not present, return "abrupt completion because of **`return`** **undefined**".
2. Evaluate *Expression*.
3. Call GetValue(Result(2)).
4. Return "abrupt completion because of **`return`** *V*", where the value *V* is Result(3).

12.10 The `with` statement

Syntax

WithStatement **:**

  `with (` *Expression* **`)`** *Statement*

Description

The **`with`** statement adds a computed object to the front of the scope chain of the current execution context, then executes a statement with this augmented scope chain, then restores the scope chain.

Semantics

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

1. Evaluate *Expression*.
2. Call GetValue(Result(1)).
3. Call ToObject(Result(2)).
4. Add Result(3) to the front of the scope chain.
5. Evaluate Statement using the augmented scope chain from step 4.
6. Remove Result(3) from the front of the scope chain.
7. Return Result(5).

NOTE No matter how control leaves the embedded *Statement*, whether normally or by some form of abrupt completion, the scope chain is always restored to its former state.

13 Function Definition

Syntax

FunctionDeclaration **:**

  `function` *Identifier* **`(`** *FormalParameterList$_{opt}$* **`)`** *Block*

FormalParameterList **:**

  Identifier
  FormalParameterList **`,`** *Identifier*

Semantics

Defines a property of the global object whose name is the *Identifier* following the **`function`** keyword and whose value is a function object with the given parameter list and statements. If the function definition is supplied text to the **`eval`** function and the calling context has an activation object, then the declared function is added to the activation object instead of to the global object. See section 10.1.3.

The production *FunctionDeclaration* **:** **`function`** *Identifier* **`( )`** *Block* is processed for function declarations as follows:

1. Create a new Function object (15.3.2.1) with no parameters, the *Block* as the body, and *Identifier* as its name.
2. Put this new Function object as the new value of the property named *Identifier* in the global object or the activation object, as appropriate (see above).

The production *FunctionDeclaration* **:** **`function`** *Identifier* **`(`** *FormalParameterList* **`)`** *Block* is processed for function declarations as follows:

1. Create a new Function object (15.3.2.1) with the parameters specified by the *FormalParameterList,* the *Block* as the body, and *Identifier* as its name.
2. Put this new Function object as the new value of the property named *Identifier* in the global object or the activation object, as appropriate (see above).

14 Program

Syntax

Program **:**

SourceElements

SourceElements **:**

SourceElement
SourceElements SourceElement

SourceElement **:**

Statement
FunctionDeclaration

Semantics

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

1. Process SourceElements for function declarations.
2. Evaluate SourceElements.
3. Return Result(2).

The production *SourceElements* **:** *SourceElement* is processed for function declarations as follows:

1. Process *SourceElement* for function declarations.

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

1. Evaluate SourceElement.
2. Return Result(1).

The production *SourceElements* **:** *SourceElements SourceElement* is processed for function declarations as follows:

1. Process *SourceElements* for function declarations.
2. Process *SourceElement* for function declarations.

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

1. Evaluate SourceElements.
2. Evaluate SourceElement.
3. If Result(2) is a value completion, return Result(2).
4. Return Result(1).

The production *SourceElement* **:** *Statement* is processed for function declarations by taking no action.

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

1. Evaluate *Statement*.
2. Return Result(1).

The production *SourceElement* **:** *FunctionDeclaration* is processed for function declarations as follows:

1. Process *FunctionDeclaration* for function declarations.

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

1. Return “normal completion”.

15 Native ECMAScript objects

There are certain built-in objects available whenever an ECMAScript program begins execution. One, the global object, is in the scope chain of the executing program. Others are accessible as initial properties of the global object.

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 section 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.

Every built-in function and every built-in constructor has the Function prototype object, which is the value of the expression **`Function.prototype`** (15.3.2.1), as the value of its internal [[Prototype]] property, except the Function prototype object itself.

Every built-in prototype object has the Object prototype object, which is the value of the expression **`Object.prototype`** (15.2.3.1), as the value of its internal [[Prototype]] property, except the Object prototype object itself. Every native prototype object associated with a program-created function also has the Object prototype object as the value of its internal [[Prototype]] property.

None of the built-in functions described in this section shall implement the internal [[Construct]] method unless otherwise specified in the description of a particular function. None of the built-in functions described in this section shall initially have a **`prototype`** property unless otherwise specified in the description of a particular function. Every built-in function object described in this section—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 number of named arguments shown in the section heading for the function description; for example, the function object that is the initial value of the **`indexOf`** property of the String prototype object is described under the section heading “indexOf(searchString, position)” which shows the two named arguments *searchString* and *position*; therefore the value of the **`length`** property of that function object is **`2`**. Sometimes the same function object is described under more than one heading to emphasise its different behaviours when given different numbers of actual arguments; in such a case, unless otherwise specified, the **`length`** value is the largest number of arguments shown in any applicable section heading. For example, the function object that is the initial value of the **`Object`** property of the global object is described under four separate headings: as a function of one argument (section 15.2.1.1), as a function of zero arguments (section 15.2.1.2), as a constructor of one argument (15.2.2.1), and as a constructor of zero arguments (15.2.2.2). The largest number of arguments described is 1, so the value of the **`length`** property of that function object is **`1`**.

In every case, a **`length`** property of a built-in function object described in this section has the attributes { ReadOnly, DontDelete, DontEnum } (and no others). Every other property described in this section has the attribute { DontEnum } (and no others) unless otherwise specified.

15.1 The Global Object

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

The value of the [[Prototype]] property of the global object is implementation-dependent.

15.1.1 Value properties of the Global Object

15.1.1.1 NaN

The initial value of **`NaN`** is **NaN**.

15.1.1.2 Infinity

The initial value of **`Infinity`** is +∞.

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 *x* is not a string value, return *x*.
2. Parse *x* as an ECMAScript *Program*. If the parse fails, generate a runtime error.
3. Evaluate the program from step 2.
4. If Result(3) is "normal completion after value *V*", return the value *V*.
5. Return undefined.

If value of the **`eval`** property is used in any way other than a direct call (that is, other than by the explicit use of its name as an *Identifier* which is the *MemberExpression* in a *CallExpression*), or if the **`eval`** property is assigned to, a runtime error may be generated.

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*.

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

1. Call ToString(*string*).
2. Compute a substring of Result(1) consisting of the leftmost character that is not a *StrWhiteSpaceChar* and all characters to the right of that character. (In other words, remove leading whitespace.)
3. Let *sign* be 1.
4. If Result(2) is not empty and the first character of Result(2) is a minus sign **`-`**, let *sign* be −1.
5. If Result(2) is not empty and the first character of Result(2) is a plus sign **`+`** or a minus sign **`-`**, then Result(5) is the substring of Result(2) produced by removing the first character; otherwise, Result(5) is Result(2).
6. If the *radix* argument is not supplied, go to step 12.
7. Call ToInt32(*radix*).
8. If Result(7) is zero, go to step 12; otherwise, if Result(7) < 2 or Result(7) > 36, return **NaN**.
9. Let *R* be Result(7).
10. If *R* = 16 and the length of Result(5) is at least 2 and the first two characters of Result(5) are either "**`0x`**" or "**`0X`**", let *S* be the substring of Result(5) consisting of all but the first two characters; otherwise, let *S* be Result(5).
11. Go to step 22.
12. If Result(5) is empty or the first character of Result(5) is not **`0`**, go to step 20.
13. If the length of Result(5) is at least 2 and the second character of Result(5) is **`x`** or **`X`**, go to step 17.
14. Let *R* be 8.
15. Let *S* be Result(5).
16. Go to step 22.
17. Let *R* be 16.
18. Let *S* be the substring of Result(5) consisting of all but the first two characters.
19. Go to step 22.
20. Let *R* be 10.
21. Let *S* be Result(5).
22. If *S* contains any character that is not a radix-*R* digit, then let *Z* be the substring of *S* consisting of all characters to the left of the leftmost such character; otherwise, let *Z* be *S*.
23. If Z is empty, return **NaN**.
24. Compute the mathematical integer value that is represented by *Z* in radix-*R* notation. (But if *R* is 10 and *Z* contains more than 20 significant digits, every 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 Result(24) may be an implementation-dependent approximation to the mathematical integer value that is represented by *Z* in radix-*R* notation.)
25. Compute the number value for Result(24).
26. Return *sign* · Result(25).

Note that **`parseInt`** may interpret only a leading portion of the 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. Call ToString(*string*).
2. Compute a substring of Result(1) consisting of the leftmost character that is not a *StrWhiteSpaceChar* and all characters to the right of that character.(In other words, remove leading whitespace.)
3. If neither Result(2) nor any prefix of Result(2) satisfies the syntax of a *StrDecimalLiteral* (see 9.3.1), return **NaN**.
4. Compute the longest prefix of Result(2), which might be Result(2) itself, which satisfies the syntax of a *StrDecimalLiteral*.
5. Return the number value for the MV of Result(4).

Note that **`parseFloat`** may interpret only a leading portion of the 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 escape(string)

The **`escape`** function computes a new version of a string value in which certain characters have been replaced by a hexadecimal escape sequence.

For those characters being replaced whose Unicode encoding is **`0xFF`** or less, a two-digit escape sequence of the form **`%`***xx* is used. For those characters being replaced whose Unicode encoding is greater than **`0xFF`**, a four-digit escape sequence of the form **`%u`***xxxx* is used

When the **`escape`** function is called with one argument *string*, the following steps are taken:

1. Call ToString(*string*).
2. Compute the number of characters in Result(1).
3. Let *R* be the empty string.
4. Let *k* be 0.
5. If *k* equals Result(2), return *R*.
6. Get the character at position *k* within Result(1).
7. If Result(6) is one of the 69 nonblank ISO/IEC 646 IRV characters **`ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz 0123456789 @*_+-./`**, go to step 14.
8. Compute the 16-bit unsigned integer that is the Unicode character encoding of Result(6).
9. If Result(8), is less than 256, go to step 12.
10. Let *S* be a string containing six characters "**`%u`***wxyz*" where *wxyz* are four hexadecimal digits encoding the value of Result(8).
11. Go to step 15.
12. Let *S* be a string containing three characters "**`%`***xy*" where *xy* are two hexadecimal digits encoding the value of Result(8).
13. Go to step 15.
14. Let *S* be a string containing the single character Result(6).
15. Let *R* be a new string value computed by concatenating the previous value of *R* and *S*.
16. Increase *k* by 1.
17. Go to step 5.

NOTE The encoding is partly based on the encoding described in RFC1738, but the entire encoding specified in this standard is described above without regard to the contents of RFC1738.

15.1.2.5 unescape(string)

The **unescape** function computes a new version of a string value in which each escape sequences of the sort that might be introduced by the **escape** function is replaced with the character that it represents.

When the **unescape** function is called with one argument *string*, the following steps are taken:

1. Call ToString(*string*).
2. Compute the number of characters in Result(1).
3. Let *R* be the empty string.
4. Let *k* be 0.
5. If *k* equals Result(2), return *R*.
6. Let *c* be the character at position *k* within Result(1).
7. If *c* is not **%**, go to step 18.
8. If *k* is greater than Result(2)−6, go to step 14.
9. If the character at position *k*+1 within result(1) is not **u**, go to step 14.
10. If the four characters at positions *k*+2, *k*+3, *k*+4, and *k*+5 within Result(1) are not all hexadecimal digits, go to step 14.
11. Let *c* be the character whose Unicode encoding is the integer represented by the four hexadecimal digits at positions *k*+2, *k*+3, *k*+4, and *k*+5 within Result(1).
12. Increase k by 5.
13. Go to step 18.
14. If *k* is greater than Result(2)−3, go to step 18.
15. If the two characters at positions *k*+1 and *k*+2 within Result(1) are not both hexadecimal digits, go to step 18.
16. Let *c* be the character whose Unicode encoding is the integer represented by two zeroes plus the two hexadecimal digits at positions *k*+1 and *k*+2 within Result(1).
17. Increase *k* by 2.
18. Let *R* be a new string value computed by concatenating the previous value of *R* and *c*.
19. Increase *k* by 1.
20. Go to step 5.

15.1.2.6 isNaN(number)

Applies ToNumber to its argument, then returns **true** if the result is **NaN**, and otherwise returns **false**.

15.1.2.7 isFinite(number)

Applies ToNumber to its argument, then returns **false** if the result is **NaN**, **+∞**, or **−∞**, and otherwise returns **true**.

15.1.3 Constructor Properties of the Global Object

15.1.3.1 Object(. . .)

See sections 15.2.1 and 15.2.2.

15.1.3.2 Function(. . .)

See sections 15.3.1 and 15.3.2.

15.1.3.3 Array(. . .)

See sections 15.4.1 and 15.4.2.

15.1.3.4 String(. . .)

See sections 15.5.1 and 15.5.2.

15.1.3.5 Boolean(. . .)

See sections 15.6.1 and 15.6.2.

15.1.3.6 Number(. . .)

See sections 15.7.1 and 15.7.2.

15.1.3.7 Date(. . .)

See section 15.9.2.

15.1.4 Other Properties of the Global Object

15.1.4.1 Math

See section 15.8.

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 one argument *value*, the following steps are taken:

1. If the *value* is **null** or **undefined**, create and return a new object with no properties (other than internal properties) exactly as if the object constructor had been called on that same value (15.2.2.1).
2. Return ToObject(*value*).

15.2.1.2 Object()

When the `Object` function is called with no arguments, the following step is taken:

1. Create and return a new object with no properties (other than internal properties) exactly if the object constructor had been called with no argument (15.2.2.2).

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 one argument *value*, the following steps are taken:

1. If the type of the *value* is not Object, go to step 4.
2. If the *value* is a native ECMAScript object, do not create a new object but simply return *value*.
3. 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.
4. If the type of the value is String, return ToObject(*value*).
5. If the type of the *value* is Boolean, return ToObject(*value*).
6. If the type of the *value* is Number, return ToObject(*value*).
7. (The type of the *value* must be Null or Undefined.) Create a new native ECMAScript object.
   The [[Prototype]] property of the newly constructed object is set to the Object prototype object.
   The [[Class]] property of the newly constructed object is set to `"Object"`.
   The newly constructed object has no [[Value]] property.
   Return the newly created native object.

15.2.2.2 new Object()

When the `Object` constructor is called with no argument, the following step is taken:

1. Create a new native ECMAScript object.
   The [[Prototype]] property of the newly constructed object is set to the Object prototype object.
   The [[Class]] property of the newly constructed object is set to `"Object"`.
   The newly constructed object has no [[Value]] property.
   Return the newly created native object.

15.2.3 Properties of the Object Constructor

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

Besides the internal [[Call]] and [[Construct]] properties and the `length` property, the Object constructor has the following properties:

15.2.3.1 Object.prototype

The initial value of `Object.prototype` is the built-in Object prototype object (15.2.4).

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.2.4 Properties of the Object Prototype Object

The value of the internal [[Prototype]] property of the Object prototype object is **null**.

15.2.4.1 Object.prototype.constructor

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

15.2.4.2 Object.prototype.toString()

When the `toString` method is called, the following steps are taken:

1. Get the [[Class]] property of this object.
2. Compute a string value by concatenating the three strings `"[object "`, Result(1), and `"]"`.
3. Return Result(2).

15.2.4.3 Object.prototype.valueOf()

As a rule, the valueOf method for an object simply returns the object; but if the object is a “wrapper” for a host object, as may perhaps be created by the Object constructor (see section 15.2.2.1), then the contained host object should be returned.

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 exactly if the function constructor had been called 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 *P* be the empty string.
2. If no arguments were given, let *body* be the empty string and go to step 13.
3. If one argument was given, let *body* be that argument and go to step 13.
4. Let Result(4) be the first argument.
5. Let *P* be ToString(Result(4)).
6. Let *k* be 2.

7. If *k* equals the number of arguments, let *body* be the *k*'th argument and go to step 13.
8. Let Result(8) be the *k*'th argument.
9. Call ToString(Result(8)).
10. Let *P* be the result of concatenating the previous value of *P*, the string **`","`** (a comma), and Result(9).
11. Increase *k* by 1.
12. Go to step 7.
13. Call ToString(*body*).
14. Let *F* be the newly constructed Function object.
15. The [[Class]] property of *F* is set to **`"Function"`**.
16. The [[Prototype]] property of *F* is set to the original Function prototype object, the one that is the initial value of **`Function.prototype`** (15.3.3.1).
17. Set the [[Call]] property of *F* to a method such that, when it is invoked, the executable function will be invoked whose formal parameters are specified by *P* and whose body is specified by Result(13). The string value *P* must be parsable as a *FormalParameterList*$_{opt}$; the string value result(13) must be parsable as a *StatementList*$_{opt}$. (Note that both *P* and Result(13) may contain whitespace, line terminators, and comments.) However, if either *P* or Result(13) is syntactically incorrect, or otherwise cannot be interpreted as part of a correct ECMAScript function definition, then the [[Call]] property of *F* is not set and a runtime error is generated..
18. Set the [[Construct]] property of *F* to a method that, when it is invoked, constructs a new object whose [[Prototype]] property is equal to the value of *F*.**`prototype`** at the time the [[Construct]] method is invoked (but if this value is not an object then the value of **`Object.prototype`** is used), then invokes *F* as a function (using its [[Call]] property) with the new object as the **`this`** value and the arguments given to the [[Construct]] method as the arguments. If the result of invoking the [[Call]] method is an object, that object becomes the result of the invocation of the [[Construct]] method; otherwise the new object becomes the result of the invocation of the [[Construct]] method.
19. If the **`toString`** method of *F* is later invoked, it will use "**`anonymous`**" as the name of the function in rendering the function as a string.
20. Compute, as an integer number value of positive sign, the number of formal parameters that resulted from the parse of *P* as a *FormalParameterList*$_{opt}$.
21. The **`length`** property of *F* is set to Result(20). This property is given attributes { DontDelete, DontEnum, ReadOnly }.
22. Create a new object as if by the expression **`new Object()`**.
23. The **`prototype`** property of *F* is set to Result(22). This property is given attributes { DontEnum }.
24. The **`constructor`** property of Result(22) is set to *F*. This property is given attributes { DontEnum }.
25. Return *F*.

Note that 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")
```

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

15.3.3 Properties of the Function Constructor

15.3.3.1 Function.prototype

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

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.3.3.2 Function.length

The **`length`** property is **`1`**. (Of course, the Function constructor accepts more than one argument, because it accepts a variable number of arguments.)

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 internal [[Prototype]] property of the Function prototype object is the Object prototype object (15.3.2.1).

It is a function with an “empty body”; if it is invoked, it merely returns **undefined**.

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

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 whitespace, line terminators, and semicolons within the representation string is implementation-dependent.

The **`toString`** function is not generic; it generates a runtime error if its **`this`** value is not a Function object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.3.5 Properties of Function Instances

Every function instance has a [[Call]] property and a [[Construct]] property.

15.3.5.1 length

The value of the **`length`** property is usually 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.

15.3.5.2 prototype

The value of the **`prototype`** property is used to initialise the internal [[Prototype]] property of a newly created object before the Function object is invoked as a constructor for that newly created object.

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 $2^{32}-1$. Every Array object has a **`length`** property whose value is always an integer with positive sign and less than 2^{32}. It is always the case that 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 properties of the Array object itself and is unaffected by **`length`** or array index properties that may be inherited from its prototype.

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(item0, item1, . . .)

An array is created and returned as if by the expression **`new Array (`***item0*, *item1*, . . .**`)`**.

15.4.1.2 Array(len)

An array is created and returned as if by the expression **`new Array (`***len***`)`**.

15.4.1.3 Array()

An array is created and returned as if by the expression **`new Array ()`**.

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 two or more arguments.

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

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*; the **`1`** property of the newly constructed object is set to *item1*; 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`**.

15.4.2.2 new Array(len)

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

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 runtime error is generated.

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*.

15.4.2.3 new Array()

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

The **`length`** property of the newly constructed object is set to **`+0`**.

15.4.3 Properties of the Array Constructor

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

Besides the internal [[Call]] and [[Construct]] properties, the Array constructor has the following properties:

15.4.3.1 Array.prototype

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

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.4.3.2 Array.length

The **`length`** property is **`1`**. (Of course, the Array constructor accepts more than one argument, because it accepts a variable number of arguments.)

15.4.4 Properties of the Array Prototype Object

The value of the internal [[Prototype]] property of the Array prototype object is the Object prototype object (15.2.3.1).

Note that the Array prototype object is itself an array; it has a **`length`** property (whose initial value is **`+0`**) and the special internal [[Put]] method described in section 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 **`this`** to refer to an object for which the value of the internal [[Class]] property is not **`"Array"`**.

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

15.4.4.1 Array.prototype.constructor

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

15.4.4.2 Array.prototype.toString()

The elements of this object are converted to strings, and these strings are then concatenated, separated by comma characters. The result is the same as if the built-in **`join`** method were invoked for this object with no argument.

15.4.4.3 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.

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

```
1. Call the [[Get]] method of this object with argument "length".
2. Call ToUint32(Result(1)).
3. If separator is not supplied, let separator be the single-character string ",".
4. Call ToString(separator).
5. If Result(2) is zero, return the empty string.
6. Call the [[Get]] method of this object with argument "0".
7. If Result(6) is undefined or null, use the empty string; otherwise, call ToString(Result(6)).
8. Let R be Result(7).
9. Let k be 1.
10.If k equals Result(2), return R.
11.Let S be a string value produced by concatenating R and Result(4).
12.Call the [[Get]] method of this object with argument ToString(k).
13.If Result(12) is undefined or null, use the empty string; otherwise, call ToString(Result(12)).
14.Let R be a string value produced by concatenating S and Result(13).
15.Increase k by 1.
16.Go to step 10.
```

Note that 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.

15.4.4.4 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. Call the [[Get]] method of this object with argument "length".
2. Call ToUint32(Result(1)).
3. Compute floor(Result(2)/2).
4. Let k be 0.
5. If k equals Result(3), return this object.
6. Compute Result(2)–k–1.
7. Call ToString(k).
8. Call ToString(Result(6)).
```

9. Call the [[Get]] method of this object with argument Result(7).
10. Call the [[Get]] method of this object with argument Result(8).
11. If this object has a property named by Result(8), go to step 12; but if this object has no property named by Result(8), then go to either step 12 or step 14, depending on the implementation.
12. Call the [[Put]] method of this object with arguments Result(7) and Result(10).
13. Go to step 15.
14. Call the [[Delete]] method on this object, providing Result(7) as the name of the property to delete.
15. If this object has a property named by Result(7), go to step 16; but if this object has no property named by Result(7), then go to either step 16 or step 18, depending on the implementation.
16. Call the [[Put]] method of this object with arguments Result(8) and Result(9).
17. Go to step 19.
18. Call the [[Delete]] method on this object, providing Result(8) as the name of the property to delete.
19. Increase *k* by 1.
20. Go to step 5.

Note that 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.

15.4.4.5 Array.prototype.sort(comparefn)

The elements of this array are sorted. The sort is not necessarily stable. If *comparefn* is supplied, 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 $x > y$.

If *comparefn* is supplied and is not a consistent comparison function for the elements of this array (see below), the result is implementation-defined. Otherwise the following steps are taken

1. Call the [[Get]] method of this object with argument "length".
2. Call ToUint32(Result(1)).
3. Perform an implementation-dependent sequence of calls to the [[Get]] , [[Put]], and [[Delete]] methods of this object and to SortCompare (described below), where the first argument for each call to [[Get]], [[Put]], or [[Delete]] is a nonnegative integer less than Result(2) and where the arguments for calls to SortCompare are results of previous calls to the [[Get]] method.
4. Return this object.

The returned object must have the following two properties.

1) There must be some mathematical permutation π of the nonnegative integers less than Result(2), such that for every nonnegative integer *j* less than Result(2), 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*)**`]`** either does not exist or exists with value **undefined**.

2) Then for all nonnegative integers *j* and *k*, each less than Result(2), if **`old[`***j***`]`** compares less than **`old[`***k***`]`** (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]] method of this object with argument *j* before this function is executed, and the notation **`new[`***j***`]`** to refer to the hypothetical result of calling the [[Get]] method of this object with argument *j* after this function has been executed.

A function is a consistent comparison function for a set of values if (a) for any two of those values (possibly the same value) considered as an ordered pair, it always returns the same value when given that pair of values as its two arguments, and the result of applying ToNumber to this value is not **NaN**; (b) when considered as a relation, where the pair (*x*, *y*) is considered to be in the relation if and only if applying the function to *x* and *y* and then applying ToNumber to the result produces a negative value, this relation is a partial order; and (c) when considered as a different relation, where the pair (*x*, *y*) is considered to be in the relation if and only if applying the function to *x* and *y* and then applying ToNumber to the result produces a zero value (of either sign), this relation is an equivalence relation. In this context, the phrase “*x* compares less than *y*” means applying Result(2) to *x* and *y* and then applying ToNumber to the result produces a negative value.

When the SortCompare operator is called with two arguments *x* and *y*, the following steps are taken:

1. If *x* and *y* are both **undefined**, return **+0**.
2. If *x* is **undefined**, return 1.
3. If *y* is **undefined**, return −1.
4. If the argument *comparefn* was not provided in the call to **`sort`**, go to step 7.
5. Call *comparefn* with arguments *x* and *y*.
6. Return Result(5).
7. Call ToString(*x*).
8. Call ToString(*y*).
9. If Result(7) < Result(8), return −1.
10. If Result(7) > Result(8), return 1.
11. Return **+0**.

Note that, because **undefined** always compares greater than any other value, undefined and non-existent property values always sort to the end of the result. It is implementation-dependent whether or not such properties will exist or not at the end of the array when the sort is concluded.

Note that 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 .

15.4.5 Properties of Array Instances

Array instances inherit properties from the Array prototype object and also have the following properties.

15.4.5.1 [[Put]](P, V)

Array objects use a variation of the [[Put]] method used for other native ECMAScript objects (section 8.6.2.2).

Assume *A* is an Array object and *P* is a string.

When the [[Put]] method of *A* is called with property *P* and value *V*, the following steps are taken:

1. Call the [[CanPut]] method of *A* with name P.
2. If Result(1) is false, return.
3. If *A* doesn’t have a property with name *P*, go to step 7.
4. If P is **`"length"`**, go to step 12.
5. Set the value of property *P* of *A* to *V*.
6. Go to step 8.
7. Create a property with name *P*, set its value to *V* and give it empty attributes.
8. If *P* is not an array index, return.
9. If ToUint32(*P*) is less than the value of the **`length`** property of *A*, then return.
10. Change (or set) the value of the **`length`** property of *A* to ToUint32(*P*)+1.
11. Return.
12. Compute ToUint32(*V*).
13. If Result(12) is not equal to ToInteger(*V*), generate a runtime error.
14. For every integer *k* that is less than the value of the **`length`** property of *A* but not less than Result(12), if *A* itself has a property (not an inherited property) named ToString(*k*), then delete that property.
15. Set the value of property *P* of *A* to Result(12).
16. Return.

15.4.5.2 length

The **`length`** property of this Array object is always numerically greater than the name of every property whose name is an array index.

The **`length`** property has the attributes { DontEnum, DontDelete }.

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).

15.5.1.2 String()

Returns the empty string **`""`**.

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]] property of the newly constructed object is set to the original String prototype object, the one that is the initial value of **`String.prototype`** (15.5.3.1).

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

The [[Value]] property of the newly constructed object is set to ToString(value).

15.5.2.2 new String()

The [[Prototype]] property of the newly constructed object is set to the original String prototype object, the one that is the initial value of **`String.prototype`** (15.5.3.1).

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

The [[Value]] property of the newly constructed object is set to the empty string.

15.5.3 Properties of the String Constructor

The value of the internal [[Prototype]] property of the String constructor is the Function prototype object.

Besides the internal [[Call]] and [[Construct]] properties and the **`length`** property, the String constructor has the following properties:

15.5.3.1 String.prototype

The initial value of **`String.prototype`** is the built-in String prototype object (15.5.4).

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

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 (section 9.7) and regarding the resulting 16-bit integer as the Unicode code point encoding of a character. If no arguments are supplied, the result is the empty string.

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 internal [[Prototype]] property of the String prototype object is the Object prototype object (15.2.3.1).

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 generates a runtime error if its **`this`** value is not 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 generates a runtime error if its **`this`** value is not a 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 ToString, giving it the **`this`** value as its argument.
2. Call ToInteger(*pos*).
3. Compute the number of characters in Result(1).
4. If Result(2) is less than 0 or is not less than Result(3), return the empty string.
5. Return a string of length 1, containing one character from Result(1), namely the character at position Result(2), where the first (leftmost) character in Result(1) 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 Unicode code point encoding 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 ToString, giving it the **`this`** value as its argument.
2. Call ToInteger(*pos*).
3. Compute the number of characters in Result(1).
4. If Result(2) is less than 0 or is not less than Result(3), return **NaN**.
5. Return a value of Number type, of positive sign, whose magnitude is the Unicode encoding of one character from Result(1), namely the character at position Result(2), where the first (leftmost) character in Result(1) 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.indexOf(searchString, position)

If the given searchString appears as a substring of the result of converting this object to a string, at one or more positions that are at or to the right of the specified position, then the index of the leftmost such position is returned; otherwise, **`-1`** is returned. If position is undefined or not supplied, 0 is assumed, so as to search all of the string.

When the **`indexOf`** method is called with two arguments *searchString* and *position*, the following steps are taken:

1. Call ToString, giving it the **`this`** value as its argument.
2. Call ToString(*searchString*).
3. Call ToInteger(*position*). (If *position* is **undefined** or not supplied, this step produces the value **`0`**).
4. Compute the number of characters in Result(1).

5. Compute min(max(Result(3), 0), Result(4)).
6. Compute the number of characters in the string that is Result(2).
7. Compute the smallest possible integer *k* not smaller than Result(5) such that *k*+Result(6) is not greater than Result(4), and for all nonnegative integers *j* less than Result(6), the character at position *k*+*j* of Result(1) is the same as the character at position *j* of Result(2); but if there is no such integer *k*, then compute the value **`-1`**.
8. Return Result(7).

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.

15.5.4.7 String.prototype.lastIndexOf(searchString, position)

If the given searchString appears as a substring of the result of converting this object to a string, at one or more positions that are at or to the left of the specified position, then the index of the rightmost such position is returned; otherwise, **`-1`** is returned. If position is undefined or not supplied, the length of the string value is assumed, so as to search all of the string.

When the **`lastIndexOf`** method is called with two arguments *searchString* and *position*, the following steps are taken:

1. Call ToString, giving it the **`this`** value as its argument.
2. Call ToString(*searchString*).
3. Call ToNumber(*position*). (If *position* is **undefined** or not supplied, this step produces the value **NaN**).
4. If Result(3) is **NaN**, use +∞; otherwise, call ToInteger(Result(3)).
5. Compute the number of characters in Result(1).
6. Compute min(max(Result(4), 0), Result(5)).
7. Compute the number of characters in the string that is Result(2).
8. Compute the largest possible integer *k* not larger than Result(6) such that *k*+Result(7) is not greater than Result(5), and for all nonnegative integers *j* less than Result(7), the character at position *k*+*j* of Result(1) is the same as the character at position *j* of Result(2); but if there is no such integer *k*, then compute the value **`-1`**.
9. Return Result(8).

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.

15.5.4.8 String.prototype.split(separator)

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 the given separator; these occurrences are not part of any substring in the returned array, but serve to divide up the string value. The separator may be a string of any length.

As a special case, if the 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 the separator is not supplied, then the result array contains just one string, which is the string.

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

1. Call ToString, giving it the **`this`** value as its argument.
2. Create a new Array object of length **`0`** and call it *A*.
3. If *separator* is not supplied, call the [[Put]] method of *A* with **`0`** and Result(1) as arguments, and then return *A*.
4. Call ToString(*separator*).
5. Compute the number of characters in Result(1).
6. Compute the number of characters in the string that is Result(4).
7. Let *p* be **`0`**.
8. If Result(6) is zero (the separator string is empty), go to step 17.

9. Compute the smallest possible integer *k* not smaller than *p* such that *k*+Result(6) is not greater than Result(5), and for all nonnegative integers *j* less than Result(6), the character at position *k*+*j* of Result(1) is the same as the character at position *j* of Result(2); but if there is no such integer *k*, then go to step 14.
10. Compute a string value equal to the substring of Result(1), consisting of the characters at positions *p* through *k*−1, inclusive.
11. Call the [[Put]] method of *A* with *A*`.length` and Result(10) as arguments.
12. Let *p* be *k*+Result(6).
13. Go to step 9.
14. Compute a string value equal to the substring of Result(1), consisting of the characters from position *p* to the end of Result(1).
15. Call the [[Put]] method of *A* with *A*`.length` and Result(14) as arguments.
16. Return *A*.
17. If p equals Result(5), return *A*.
18. Compute a string value equal to the substring of Result(1), consisting of the single character at position *p*.
19. Call the [[Put]] method of *A* with *A*`.length` and Result(18) as arguments.
20. Increase *p* by 1.
21. Go to step 17.

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

15.5.4.9 String.prototype.substring(start)

Returns a substring of the result of converting this object to a string, starting from character position *start* and running to the end of the string. The result is a string value, not a string object.

If the argument is **NaN** or negative, it is replaced with zero; if the argument is larger than the length of the string, it is replaced with the length of the string.

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

1. Call ToString, giving it the `this` value as its argument.
2. Call ToInteger(*start*).
3. Compute the number of characters in Result(1).
4. Compute min(max(Result(2), 0), Result(3)).
5. Return a string whose length is the difference between Result(3) and Result(4), containing characters from Result(1), namely the characters with indices Result(4) through Result(3)−1, in ascending order.

15.5.4.10 String.prototype.substring(start, end)

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. 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.

When the `substring` method is called with two arguments *start* and *end*, the following steps are taken:

1. Call ToString, giving it the `this` value as its argument.
2. Call ToInteger(*start*).
3. Call ToInteger (*end*).
4. Compute the number of characters in Result(1).
5. Compute min(max(Result(2), 0), Result(4)).
6. Compute min(max(Result(3), 0), Result(4)).
7. Compute min(Result(5), Result(6)).
8. Compute max(Result(5), Result(6)).

9. Return a string whose length is the difference between Result(8) and Result(7), containing characters from Result(1), namely the characters with indices Result(7) through Result(8)−1, in ascending order.

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.

15.5.4.11 String.prototype.toLowerCase

Returns a string equal in length to the length of the result of converting this object to a string. The result is a string value, not a string object.

Every character of the result is equal to the corresponding character of the string, unless that character has a Unicode 2.0 lowercase equivalent, in which case the lowercase equivalent is used instead. (The canonical Unicode 2.0 case mapping shall be used, which does not depend on implementation or locale.)

Note that 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.

15.5.4.12 String.prototype.toUpperCase

Returns a string equal in length to the length of the result of converting this object to a string. The result is a string value, not a string object.

Every character of the result is equal to the corresponding character of the string, unless that character has a Unicode 2.0 uppercase equivalent, in which case the uppercase equivalent is used instead. (The canonical Unicode 2.0 case mapping shall be used, which does not depend on implementation or locale.)

Note that 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.

15.5.5 Properties of String Instances

String instances inherit properties from the String prototype object and also have a [[Value]] property and a **`length`** property.

The [[Value]] property is the string value represented by this string object.

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 { DontEnum, DontDelete, ReadOnly }.

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).

15.6.1.2 Boolean()

Returns **false**.

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]] 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]] property of the newly constructed boolean object is set to **`"Boolean"`**.

The [[Value]] property of the newly constructed boolean object is set to ToBoolean(value).

15.6.2.2 new Boolean()

The [[Prototype]] 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]] property of the newly constructed boolean object is set to `"Boolean"`.

The [[Value]] property of the newly constructed boolean object is set to **false**.

15.6.3 Properties of the Boolean Constructor

The value of the internal [[Prototype]] property of the Boolean constructor is the Function prototype object.

Besides the internal [[Call]] and [[Construct]] properties and the `length` property, the Boolean constructor has the following property:

15.6.3.1 Boolean.prototype

The initial value of `Boolean.prototype` is the built-in Boolean prototype object (15.6.4).

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

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 internal [[Prototype]] property of the Boolean prototype object is the Object prototype object (15.2.3.1).

In following descriptions of functions that are properties of the Boolean prototype object, the phrase “this boolean object” refers to the object that is the `this` value for the invocation of the function; it is a runtime error if `this` does not refer to an object for which the value of the internal [[Class]] property is `"Boolean"`. Also, the phrase “this boolean value” refers to the boolean value represented by this boolean object, that is, the value of the internal [[Value]] property of this boolean object.

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()

If this boolean value is **true**, then the string `"true"` is returned. Otherwise, this boolean value must be **false**, and the string `"false"` is returned.

The `toString` function is not generic; it generates a runtime error if its `this` value is not a boolean object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.6.4.3 Boolean.prototype.valueOf()

Returns this boolean value.

The `valueOf` function is not generic; it generates a runtime error if its `this` value is not a boolean object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.6.5 Properties of Boolean Instances

Boolean instances have no special properties beyond those inherited from the Boolean prototype 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).

15.7.1.2 Number()

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]] 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]] property of the newly constructed object is set to **`"Number"`**.

The [[Value]] property of the newly constructed object is set to ToNumber(value).

15.7.2.2 new Number()

The [[Prototype]] 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]] property of the newly constructed object is set to **`"Number"`**.

The [[Value]] property of the newly constructed object is set to **`+0`**.

15.7.3 Properties of the Number Constructor

The value of the internal [[Prototype]] property of the Number constructor is the Function prototype object.

Besides the internal [[Call]] and [[Construct]] properties and the **`length`** property, the Number constructor has the following property:

15.7.3.1 Number.prototype

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

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

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 **`1.7976931348623157e308`**.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.7.3.3 Number.MIN_VALUE

The value of **`Number.MIN_VALUE`** is the smallest positive nonzero value of the number type, which is approximately **`5e-324`**.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.7.3.4 Number.NaN

The value of **`Number.NaN`** is **NaN**.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.7.3.5 Number.NEGATIVE_INFINITY

The value of **`Number.NEGATIVE_INFINITY`** is −∞.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.7.3.6 Number.POSITIVE_INFINITY

The value of **`Number.POSITIVE_INFINITY`** is +∞.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

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 internal [[Prototype]] property of the Number prototype object is the Object prototype object (15.2.3.1).

In following descriptions of functions that are properties of the Number prototype object, the phrase "this Number object" refers to the object that is the **`this`** value for the invocation of the function; it is a runtime error if **`this`** does not refer to an object for which the value of the internal [[Class]] property is **`"Number"`**. Also, the phrase "this number value" refers to the number value represented by this Number object, that is, the value of the internal [[Value]] property of this Number object.

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)

If the *radix* is the number 10 or not supplied, then this number value is given as an argument to the ToString operator; the resulting string value is returned.

If the *radix* is supplied and is an integer from 2 to 36, but not 10, the result is a string, the choice of which is implementation-dependent.

The **`toString`** function is not generic; it generates a runtime error if its **`this`** value is not a Number object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.7.4.3 Number.prototype.valueOf()

Returns this number value.

The **`valueOf`** function is not generic; it generates a runtime error if its **`this`** value is not a Number object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.7.5 Properties of Number Instances

Number instances have no special properties beyond those inherited from the Number prototype object.

15.8 The Math Object

The Math object is merely a single object that has some named properties, some of which are functions.

The value of the internal [[Prototype]] property of the Math object is the Object prototype object (15.2.3.1).

The Math object does not have a [[Construct]] 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]] 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 section 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 shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.8.1.2 LN10

The number value for the natural logarithm of 10, which is approximately **`2.302585092994046`**.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.8.1.3 LN2

The number value for the natural logarithm of 2, which is approximately **`0.6931471805599453`**.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

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`**. (Note that the value of **`Math.LOG2E`** is approximately the reciprocal of the value of **`Math.LN2`**.)

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

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`**. (Note that the value of **`Math.LOG10E`** is approximately the reciprocal of the value of **`Math.LN10`**.)

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.8.1.6 PI

The number value for π, the ratio of the circumference of a circle to its diameter, which is approximately **`3.14159265358979323846`**.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.8.1.7 SQRT1_2

The number value for the square root of 1/2, which is approximately **`0.7071067811865476`**. (Note that the value of **`Math.SQRT1_2`** is approximately the reciprocal of the value of **`Math.SQRT2`**.)

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.8.1.8 SQRT2

The number value for the square root of 2, which is approximately **`1.4142135623730951`**.

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

15.8.2 Function Properties of the Math Object

Every function listed in this section applies the ToNumber 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).

The behaviour of the functions **`acos`**, **`asin`**, **`atan`**, **`atan2`**, **`cos`**, **`exp`**, **`log`**, **`pow`**, **`sin`**, and **`sqrt`** 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 implementor 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 (`fdlibm-comment@sunpro.eng.sun.com`).

15.8.2.1 abs(x)

Returns the absolute value of its argument; in general, the result has the same magnitude as the argument but has positive sign.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **−0**, the result is **+0**.
- If the argument is −∞, the result is +∞.

15.8.2.2 acos(x)

Returns an implementation-dependent approximation to the arc cosine of the argument. The result is expressed in radians and ranges from **+0** to +π.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is greater than **`1`**, the result is **NaN**.
- If the argument is less than **`-1`**, the result is **NaN**.
- If the argument is exactly **`1`**, the result is **+0**.

15.8.2.3 asin(x)

Returns an implementation-dependent approximation to the arc sine of the argument. The result is expressed in radians and ranges from −π/2 to +π/2.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is greater than **1**, the result is **NaN**.
- If the argument is less than **-1**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **−0**, the result is **−0**.

15.8.2.4 atan(x)

Returns an implementation-dependent approximation to the arc tangent of the argument. The result is expressed in radians and ranges from −π/2 to +π/2.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **−0**, the result is **−0**.
- If the argument is +∞, the result is an implementation-dependent approximation to +π/2.
- If the argument 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 the arguments 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 argument 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 the argument and is equal to a mathematical integer. If the argument is already an integer, the result is the argument itself.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **−0**, the result is **−0**.
- If the argument is +∞, the result is +∞.
- If the argument is −∞, the result is −∞.
- If the argument 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 the argument. The argument is expressed in radians.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **1**.
- If the argument is **−0**, the result is **1**.
- If the argument is +∞, the result is **NaN**.
- If the argument is −∞, the result is **NaN**.

15.8.2.8 exp(x)

Returns an implementation-dependent approximation to the exponential function of the argument (*e* raised to the power of the argument, where *e* is the base of the natural logarithms).

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **1**.
- If the argument is **−0**, the result is **1**.
- If the argument is +∞, the result is +∞.
- If the argument is −∞, the result is **+0**.

15.8.2.9 floor(x)

Returns the greatest (closest to +∞) number value that is not greater than the argument and is equal to a mathematical integer. If the argument is already an integer, the result is the argument itself.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **−0**, the result is **−0**.
- If the argument is +∞, the result is +∞.
- If the argument is −∞, the result is −∞.
- If the argument 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 natural logarithm of the argument.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is less than **0**, the result is **NaN**.
- If the argument is **+0** or −**0**, the result is −∞.
- If the argument is **1**, the result is **+0**.
- If the argument is +∞, the result is +∞.

15.8.2.11 max(x, y)

Returns the larger of the two arguments.

- If either argument is **NaN**, the result is **NaN**.
- If `x>y`, the result is `x`.
- If `y>x`, the result is `y`.
- If `x` is +**0** and `y` is +**0**, the result is +**0**.
- If `x` is +**0** and `y` is −**0**, the result is +**0**.
- If `x` is −**0** and `y` is +**0**, the result is +**0**.
- If `x` is −**0** and `y` is −**0**, the result is −**0**.
- Otherwise `x` and `y` are identical; the result is that value.

15.8.2.12 min(x, y)

Returns the smaller of the two arguments.

- If either argument is **NaN**, the result is **NaN**.
- If `x<y`, the result is `x`.
- If `y<x`, the result is `y`.
- If `x` is +**0** and `y` is +**0**, the result is +**0**.
- If `x` is +**0** and `y` is −**0**, the result is −**0**.
- If `x` is −**0** and `y` is +**0**, the result is −**0**.
- If `x` is −**0** and `y` is −**0**, the result is −**0**.
- Otherwise `x` and `y` are identical; the result is that value.

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 the argument and is equal to a mathematical integer. If two integer number values are equally close to the argument, then the result is the number value that is closer to +∞. If the argument is already an integer, the result is the argument itself.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **−0**, the result is **−0**.
- If the argument is +∞, the result is +∞.
- If the argument is −∞, the result is −∞.
- If the argument is greater than **`0`** but less than **`0.5`**, the result is **+0**.
- If the argument is less than **`0`** but greater than or equal to **`-0.5`**, the result is −**0**.

NOTE Math.round(3.5) returns 4, but Math.round(-3.5) returns -3.

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 the argument. The argument is expressed in radians.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **−0**, the result is **−0**.
- If the argument is +∞ or −∞, the result is **NaN**.

15.8.2.17 sqrt(x)

Returns an implementation-dependent approximation to the square root of the argument.

- If the argument is **NaN**, the result is **NaN**.
- If the argument less than **`0`**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **−0**, the result is **−0**.
- If the argument is +∞, the result is +∞.

15.8.2.18 tan(x)

Returns an implementation-dependent approximation to the tangent of the argument. The argument is expressed in radians.

- If the argument is **NaN**, the result is **NaN**.
- If the argument is **+0**, the result is **+0**.
- If the argument is **−0**, the result is **−0**.
- If the argument is +∞ or −∞, the result is **NaN**.

15.9 Date Objects

15.9.1 Overview of Date Objects and Definitions of Internal Operators

A Date object contains a number indicating a particular instant in time to within a millisecond. The number may also be **NaN**, indicating that the Date object does not represent a specific instant of time.

The following sections define a number of functions for operating on time values. Note that, in every case, if any argument to such a function is **NaN**, the result will be **NaN**.

15.9.1.1 Time Range

Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC. 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 –9,007,199,254,740,991 to 9,007,199,254,740,991; 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

Day(*t*) = floor(*t* / msPerDay)

where the number of milliseconds per day is

msPerDay = 86400000

The remainder is called the time within the day:

TimeWithinDay(*t*) = *t* modulo msPerDay

15.9.1.3 Year Number

ECMAScript uses an extrapolated Gregorian system to map a day number to a year number and to determine the month and date within that year. In this system, leap years are precisely those which are (divisible by 4) and ((not divisible by 100) or (divisible by 400)). The number of days in year number *y* is therefore defined by

$$\begin{aligned}\text{DaysInYear}(y) &= 365 \ \text{ if } (y \text{ modulo } 4) \neq 0\\ &= 366 \ \text{ if } (y \text{ modulo } 4) = 0 \text{ and } (y \text{ modulo } 100) \neq 0\\ &= 365 \ \text{ if } (y \text{ modulo } 100) = 0 \text{ and } (y \text{ modulo } 400) \neq 0\\ &= 366 \ \text{ if } (y \text{ modulo } 400) = 0\end{aligned}$$

All non-leap years have 365 days with the usual number of days per month and leap years have an extra day in February. The day number of the first day of year *y* is given by:

$$\text{DayFromYear}(y) = 365\cdot(y-1970) + \text{floor}((y-1969)/4) - \text{floor}((y-1901)/100) + \text{floor}((y-1601)/400)$$

The time value of the start of a year is:

$$\text{TimeFromYear}(y) = \text{msPerDay}\cdot\text{DayFromYear}(y)$$

A time value determines a year by:

YearFromTime(*t*) = the largest integer *y* (closest to positive infinity) such that TimeFromYear(*y*) ≤ *t*

The leap-year function is 1 for a time within a leap year and otherwise is zero:

$$\begin{aligned}\text{InLeapYear}(t) &= 0 \ \text{ if DaysInYear(YearFromTime}(t)) = 365\\ &= 1 \ \text{ if DaysInYear(YearFromTime}(t)) = 366\end{aligned}$$

15.9.1.4 Month Number

Months are identified by an integer in the range 0 to 11, inclusive. The mapping MonthFromTime(*t*) from a time value *t* to a month number is defined by:

MonthFromTime(*t*)	= 0	if	0	≤ DayWithinYear(*t*) < 31
	= 1	if	31	≤ DayWithinYear (*t*) < 59+InLeapYear(*t*)
	= 2	if	59+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 90+InLeapYear(*t*)
	= 3	if	90+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 120+InLeapYear(*t*)
	= 4	if	120+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 151+InLeapYear(*t*)
	= 5	if	151+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 181+InLeapYear(*t*)
	= 6	if	181+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 212+InLeapYear(*t*)
	= 7	if	212+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 243+InLeapYear(*t*)
	= 8	if	243+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 273+InLeapYear(*t*)
	= 9	if	273+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 304+InLeapYear(*t*)
	= 10	if	304+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 334+InLeapYear(*t*)
	= 11	if	334+InLeapYear(*t*)	≤ DayWithinYear (*t*) < 365+InLeapYear(*t*)

where

DayWithinYear(*t*) = Day(*t*)−DayFromYear(YearFromTime(*t*))

A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10 specifies November; and 11 specifies December. Note that MonthFromTime(0) = 0, corresponding to Thursday, 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:

DateFromTime(*t*)	= DayWithinYear(*t*)+1	if MonthFromTime(*t*)=0
	= DayWithinYear(*t*)−30	if MonthFromTime(*t*)=1
	= DayWithinYear(*t*)−58−InLeapYear(*t*)	if MonthFromTime(*t*)=2
	= DayWithinYear(*t*)−89−InLeapYear(*t*)	if MonthFromTime(*t*)=3
	= DayWithinYear(*t*)−119−InLeapYear(*t*)	if MonthFromTime(*t*)=4
	= DayWithinYear(*t*)−150−InLeapYear(*t*)	if MonthFromTime(*t*)=5
	= DayWithinYear(*t*)−180−InLeapYear(*t*)	if MonthFromTime(*t*)=6
	= DayWithinYear(*t*)−211−InLeapYear(*t*)	if MonthFromTime(*t*)=7
	= DayWithinYear(*t*)−242−InLeapYear(*t*)	if MonthFromTime(*t*)=8
	= DayWithinYear(*t*)−272−InLeapYear(*t*)	if MonthFromTime(*t*)=9
	= DayWithinYear(*t*)−303−InLeapYear(*t*)	if MonthFromTime(*t*)=10
	= DayWithinYear(*t*)−333−InLeapYear(*t*)	if MonthFromTime(*t*)=11

15.9.1.6 Week Day

The weekday for a particular time value *t* is defined as

WeekDay(*t*) = (Day(*t*) + 4) modulo 7

A weekday value of 0 specifies Sunday; 1 specifies Monday; 2 specifies Tuesday; 3 specifies Wednesday; 4 specifies Thursday; 5 specifies Friday; and 6 specifies Saturday. Note that WeekDay(0) = 4, corresponding to Thursday, 01 January, 1970.

15.9.1.7 Local Time Zone Adjustment

An implementation of ECMAScript is expected to determine the local time zone adjustment. The local time zone adjustment is a value LocalTZA measured in milliseconds which when added to UTC represents the local *standard* time. Daylight saving time is *not* reflected by LocalTZA. 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 – TimeFromYear(YearFromTime(*t*))

(2) whether *t* is in a leap year

InLeapYear(*t*)

(3) the week day of the beginning of the year

WeekDay(TimeFromYear(YearFromTime(*t*))

and (4) the geographic location.

The implementation of ECMAScript should 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 underlying operating system 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 operating system 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 = msPerSecond · SecondsPerMinute = 60000

msPerHour = msPerMinute · MinutesPerHour = 3600000

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. Call ToInteger(*hour*).
3. Call ToInteger(*min*).
4. Call ToInteger(*sec*).
5. Call ToInteger(*ms*).

6. Compute Result(2) **`*`** msPerHour **`+`** Result(3) **`*`** msPerMinute **`+`** Result(4) **`*`** msPerSecond **`+`** Result(5), performing the arithmetic according to IEEE 754 rules (that is, as if using the ECMAScript operators **`*`** and **`+`**).
7. Return Result(6).

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. Call ToInteger(*year*).
3. Call ToInteger(*month*).
4. Call ToInteger(*date*).
5. Compute Result(2) + floor(Result(3)/12).
6. Compute Result(3) modulo 12.
7. Find a value *t* such that YearFromTime(*t*)**`==`**Result(5) and MonthFromTime(*t*)**`==`**Result(6) and DateFromTime(*t*)**`==`**1; but if this is not possible (because some argument is out of range), return **NaN**.
8. Compute Day(Result(7)) + Result(4) − 1.
9. Return Result(8).

15.9.1.13 MakeDate(day, time)

The operator MakeDate calculates a number of milliseconds from its two arguments, which must be ECMAScript number values. This operator functions as follows:

1. If *day* is not finite or *time* is not finite, return **NaN**.
2. Compute *day* · msPerDay + *time*.
3. Return Result(2).

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(Result(1)) > **`8.64e15`** (that is, $8.64 \cdot 10^{15}$), return **NaN**.
3. Return an implementation-dependent choice of either ToInteger(Result(2)) or ToInteger(Result(2)) + (+**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**.

15.9.2 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 that 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)

The arguments are accepted but are completely ignored. A string is created and returned as if by the expression **`(new Date ()).toString()`**.

15.9.2.2 Date(year, month, date, hours, minutes, seconds)

The arguments are accepted but are completely ignored. A string is created and returned as if by the expression **`(new Date ()).toString()`**.

15.9.2.3 Date(year, month, date, hours, minutes)

The arguments are accepted but are completely ignored. A string is created and returned as if by the expression **`(new Date ()).toString()`**.

15.9.2.4 Date(year, month, date, hours)

The arguments are accepted but are completely ignored. A string is created and returned as if by the expression `(new Date ()).toString()`.

15.9.2.5 Date(year, month, day)

The arguments are accepted but are completely ignored. A string is created and returned as if by the expression `(new Date ()).toString()`.

15.9.2.6 Date(year, month)

The arguments are accepted but are completely ignored. A string is created and returned as if by the expression `(new Date ()).toString()`.

15.9.2.7 Date(value)

The argument is accepted but is completely ignored. A string is created and returned as if by the expression `(new Date ()).toString()`.

15.9.2.8 Date()

A string is created and returned as if by the expression `new Date ().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)

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

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

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. Call ToNumber(*hours*).
5. Call ToNumber(*minutes*).
6. Call ToNumber(*seconds*).
7. Call ToNumber(*ms*).
8. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(8) is 1900+ToInteger(Result(1)); otherwise, Result(8) is Result(1).
9. Compute MakeDay(Result(8), Result(2), Result(3)).
10. Compute MakeTime(Result(4), Result(5), Result(6), Result(7)).
11. Compute MakeDate(Result(9), Result(10)).
12. Set the [[Value]] property of the newly constructed object to TimeClip(UTC(Result(11))).

15.9.3.2 new Date(year, month, date, hours, minutes, seconds)

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

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

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. Call ToNumber(*hours*).
5. Call ToNumber(*minutes*).
6. Call ToNumber(*seconds*).
7. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(7) is 1900+ToInteger(Result(1)); otherwise, Result(7) is Result(1).

8. Compute MakeDay(Result(7), Result(2), Result(3)).
9. Compute MakeTime(Result(4), Result(5), Result(6), 0).
10. Compute MakeDate(Result(8), Result(9)).
11. Set the [[Value]] property of the newly constructed object to TimeClip(UTC(Result(10))).

15.9.3.3 new Date(year, month, date, hours, minutes)

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

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

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. Call ToNumber(*hours*).
5. Call ToNumber(*minutes*).
6. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(6) is 1900+ToInteger(Result(1)); otherwise, Result(6) is Result(1).
7. Compute MakeDay(Result(6), Result(2), Result(3)).
8. Compute MakeTime(Result(4), Result(5), 0, 0).
9. Compute MakeDate(Result(7), Result(8)).
10. Set the [[Value]] property of the newly constructed object to TimeClip(UTC(Result(9))).

15.9.3.4 new Date(year, month, date, hours)

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

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

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. Call ToNumber(*hours*).
5. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(5) is 1900+ToInteger(Result(1)); otherwise, Result(5) is Result(1).
6. Compute MakeDay(Result(5), Result(2), Result(3)).
7. Compute MakeTime(Result(4), 0, 0, 0).
8. Compute MakeDate(Result(6), Result(7)).
9. Set the [[Value]] property of the newly constructed object to TimeClip(UTC(Result(8))).

15.9.3.5 new Date(year, month, day)

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

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

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(4) is 1900+ToInteger(Result(1)); otherwise, Result(4) is Result(1).
5. Compute MakeDay(Result(4), Result(2), Result(3)).
6. Compute MakeDate(Result(5), 0).
7. Set the [[Value]] property of the newly constructed object to TimeClip(UTC(Result(6))).

15.9.3.6 new Date(year, month)

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

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

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(3) is 1900+ToInteger(Result(1)); otherwise, Result(3) is Result(1).
4. Compute MakeDay(Result(3), Result(2), 1).
5. Compute MakeDate(Result(4), 0).
6. Set the [[Value]] property of the newly constructed object to TimeClip(UTC(Result(5))).

15.9.3.7 new Date(value)

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

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

The [[Value]] property of the newly constructed object is set as follows:

1. Call ToPrimitive(value).
2. If Type(Result(1)) is String, then go to step 5.
3. Let *V* be ToNumber(Result(1)).
4. Set the [[Value]] property of the newly constructed object to TimeClip(*V*) and return.
5. Parse Result(1) as a date, in exactly the same manner as for the **`parse`** method (); let *V* be the time value for this date.
6. Go to step 4.

15.9.3.8 new Date()

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

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

The [[Value]] property of the newly constructed object is set to the current time (UTC).

15.9.4 Properties of the Date Constructor

The value of the internal [[Prototype]] property of the Date constructor is the Function prototype object.

Besides the internal [[Call]] and [[Construct]] 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 ().

This property shall have the attributes { DontEnum, DontDelete, ReadOnly }.

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; it returns a number, the UTC time value corresponding to the date. 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.

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.toGMTString())
```

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 could not be produced in that implementation by the **`toString`** or **`toGMTString`** method.

15.9.4.3 Date.UTC(year, month, date, hours, minutes, seconds, ms)

When the **`UTC`** function is called with seven arguments, the following steps are taken:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. Call ToNumber(*hours*).
5. Call ToNumber(*minutes*).
6. Call ToNumber(*seconds*).
7. Call ToNumber(*ms*).
8. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(8) is 1900+ToInteger(Result(1)); otherwise, Result(8) is Result(1).
9. Compute MakeDay(Result(8), Result(2), Result(3)).
10. Compute MakeTime(Result(4), Result(5), Result(6), Result(7)).
11. Return TimeClip(MakeDate(Result(9), Result(10))).

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.UTC(year, month, date, hours, minutes, seconds)

When the **`UTC`** function is called with six arguments, the following steps are taken:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. Call ToNumber(*hours*).
5. Call ToNumber(*minutes*).
6. Call ToNumber(*seconds*).
7. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(7) is 1900+ToInteger(Result(1)); otherwise, Result(7) is Result(1).
8. Compute MakeDay(Result(7), Result(2), Result(3)).
9. Compute MakeTime(Result(4), Result(5), Result(6), 0).
10. Return TimeClip(MakeDate(Result(8), Result(9))).

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.

15.9.4.5 Date.UTC(year, month, date, hours, minutes)

When the **`UTC`** function is called with five arguments, the following steps are taken:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. Call ToNumber(*hours*).
5. Call ToNumber(*minutes*).
6. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(6) is 1900+ToInteger(Result(1)); otherwise, Result(6) is Result(1).
7. Compute MakeDay(Result(6), Result(2), Result(3)).
8. Compute MakeTime(Result(4), Result(5), 0, 0).
9. Return TimeClip(MakeDate(Result(7), Result(8))).

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.

15.9.4.6 Date.UTC(year, month, date, hours)

When the **`UTC`** function is called with four arguments, the following steps are taken:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. Call ToNumber(*hours*).
5. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(5) is 1900+ToInteger(Result(1)); otherwise, Result(5) is Result(1).
6. Compute MakeDay(Result(5), Result(2), Result(3)).
7. Compute MakeTime(Result(4), 0, 0, 0).
8. Return TimeClip(MakeDate(Result(6), Result(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.

15.9.4.7 Date.UTC(year, month, date)

When the **`UTC`** function is called with three arguments, the following steps are taken:

1. Call ToNumber(*year*).
2. Call ToNumber(*month*).
3. Call ToNumber(*date*).
4. If Result(1) is not **NaN** and 0 ≤ ToInteger(Result(1)) ≤ 99, Result(4) is 1900+ToInteger(Result(1)); otherwise, Result(4) is Result(1).
5. Compute MakeDay(Result(4), Result(2), Result(3)).
6. Return TimeClip(MakeDate(Result(5), 0)).

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.

15.9.4.8 Date.UTC(year, month)

The behaviour of the **`UTC`** function with two arguments is implementation-dependent.

15.9.4.9 Date.UTC(year)

The behaviour of the **`UTC`** function with one argument is implementation-dependent.

15.9.4.10 Date.UTC()

The behaviour of the **`UTC`** function with no arguments is implementation-dependent.

15.9.5 Properties of the Date Prototype Object

The Date prototype object is itself a Date object (its [[Class]] is **`"Date"`**) whose value is **NaN**.

The value of the internal [[Prototype]] property of the Date prototype object is the Object prototype object (15.2.3.1).

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; it is a runtime error if **`this`** does not refer to an object for which the value of the internal [[Class]] 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 internal [[Value]] 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 a convenient, human-readable form in the current time zone.

The **`toString`** function is not generic; it generates a runtime error if its **`this`** value is not a Date object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.9.5.3 Date.prototype.valueOf()

The **`valueOf`** function returns a number, which is this time value.

The **`valueOf`** function is not generic; it generates a runtime error if its **`this`** value is not a Date object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.9.5.4 Date.prototype.getTime()

1. If the **`this`** value is not an object whose [[Class]] property is **`"Date"`**, generate a runtime error.
2. Return this time value.

15.9.5.5 Date.prototype.getYear()

NOTE This function is not part of this specification. The function **`getFullYear`** is much to be preferred for nearly all purposes, because it avoids the "year 2000 problem." If implemented, **getYear** may follow the following rules:

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(LocalTime(*t*)) − 1900.

15.9.5.6 Date.prototype.getFullYear()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(LocalTime(*t*)).

15.9.5.7 Date.prototype.getUTCFullYear()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return YearFromTime(*t*).

15.9.5.8 Date.prototype.getMonth()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MonthFromTime(LocalTime(*t*)).

15.9.5.9 Date.prototype.getUTCMonth()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MonthFromTime(*t*).

15.9.5.10 Date.prototype.getDate()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return DateFromTime(LocalTime(*t*)).

15.9.5.11 Date.prototype.getUTCDate()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return DateFromTime(*t*).

15.9.5.12 Date.prototype.getDay()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return WeekDay(LocalTime(*t*)).

15.9.5.13 Date.prototype.getUTCDay()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.

3. Return WeekDay(*t*).

15.9.5.14 Date.prototype.getHours()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return HourFromTime(LocalTime(*t*)).

15.9.5.15 Date.prototype.getUTCHours()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return HourFromTime(*t*).

15.9.5.16 Date.prototype.getMinutes()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MinFromTime(LocalTime(*t*)).

15.9.5.17 Date.prototype.getUTCMinutes()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return MinFromTime(*t*).

15.9.5.18 Date.prototype.getSeconds()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return SecFromTime(LocalTime(*t*)).

15.9.5.19 Date.prototype.getUTCSeconds()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return SecFromTime(*t*).

15.9.5.20 Date.prototype.getMilliseconds()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return msFromTime(LocalTime(*t*)).

15.9.5.21 Date.prototype.getUTCMilliseconds()

1. Let *t* be this time value.
2. If *t* is **NaN**, return **NaN**.
3. Return msFromTime(*t*).

15.9.5.22 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.

15.9.5.23 Date.prototype.setTime(time)

1. If the **`this`** value is not a Date object, generate a runtime error.
2. Call ToNumber(*time*).
3. Call TimeClip(Result(1)).
4. Set the [[Value]] property of the **`this`** value to Result(2).
5. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.24 Date.prototype.setMilliseconds(ms)

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*ms*).
3. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), Result(2)).

4. Compute UTC(MakeDate(Day(*t*), Result(3))).
5. Set the [[Value]] property of the **`this`** value to TimeClip(Result(4)).
6. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.25 Date.prototype.setUTCMilliseconds(ms)

1. Let *t* be this time value.
2. Call ToNumber(*ms*).
3. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), SecFromTime(*t*), Result(2)).
4. Compute MakeDate(Day(*t*), Result(3)).
5. Set the [[Value]] property of the **`this`** value to TimeClip(Result(4)).
6. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.26 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. Call ToNumber(*sec*).
3. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
4. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), Result(2), Result(3)).
5. Compute UTC(MakeDate(Day(*t*), Result(4))).
6. Set the [[Value]] property of the **`this`** value to TimeClip(Result(5)).
7. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.27 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. Call ToNumber(*sec*).
3. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
4. Compute MakeTime(HourFromTime(*t*), MinFromTime(*t*), Result(2), Result(3)).
5. Compute MakeDate(Day(*t*), Result(4)).
6. Set the [[Value]] property of the **`this`** value to TimeClip(Result(5)).
7. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.28 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. Call ToNumber(*min*).
3. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
4. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
5. Compute MakeTime(HourFromTime(*t*), Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Day(*t*), Result(5))).
7. Set the [[Value]] property of the **`this`** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.29 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 behaves as if *ms* were specified with the value getUTCMilliseconds( ).

1. Let *t* be this time value.
2. Call ToNumber(*min*).
3. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
4. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
5. Compute MakeTime(HourFromTime(*t*), Result(2), Result(3), Result(4)).
6. Compute MakeDate(Day(*t*), Result(5)).
7. Set the [[Value]] property of the **`this`** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.30 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. Call ToNumber(*hour*).
3. If *min* is not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
4. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
5. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute UTC(MakeDate(Day(*t*), Result(6))).
8. Set the [[Value]] property of the **`this`** value to TimeClip(Result(7)).
9. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.31 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. Call ToNumber(*hour*).
3. If *min* is not specified, compute MinFromTime(*t*); otherwise, call ToNumber(*min*).
4. If *sec* is not specified, compute SecFromTime(*t*); otherwise, call ToNumber(*sec*).
5. If *ms* is not specified, compute msFromTime(*t*); otherwise, call ToNumber(*ms*).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute MakeDate(Day(*t*), Result(6)).
8. Set the [[Value]] property of the **`this`** value to TimeClip(Result(7)).
9. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.32 Date.prototype.setDate(date)

1. Let *t* be the result of LocalTime(this time value).
2. Call ToNumber(*date*).
3. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
4. Compute UTC(MakeDate(Result(3), TimeWithinDay(*t*))).
5. Set the [[Value]] property of the **`this`** value to TimeClip(Result(4)).
6. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.33 Date.prototype.setUTCDate(date)

1. Let *t* be this time value.
2. Call ToNumber(*date*).
3. Compute MakeDay(YearFromTime(*t*), MonthFromTime(*t*), Result(2)).
4. Compute MakeDate(Result(3), TimeWithinDay(*t*)).
5. Set the [[Value]] property of the **`this`** value to TimeClip(Result(4)).
6. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.34 Date.prototype.setMonth(mon [, 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. Call ToNumber(*mon*).
3. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
4. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).
5. Compute UTC(MakeDate(Result(4), TimeWithinDay(*t*))).
6. Set the [[Value]] property of the **`this`** value to TimeClip(Result(5)).
7. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.35 Date.prototype.setUTCMonth(mon [, 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. Call ToNumber(*mon*).
3. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
4. Compute MakeDay(YearFromTime(*t*), Result(2), Result(3)).
5. Compute MakeDate(Result(4), TimeWithinDay(*t*)).
6. Set the [[Value]] property of the **`this`** value to TimeClip(Result(5)).
7. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.36 Date.prototype.setFullYear(year [, mon [, date]])

If *mon* is not specified, this behaves as if *mon* 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. Call ToNumber(*year*).
3. If *mon* is not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*mon*).
4. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
5. Compute MakeDay(Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Result(5), TimeWithinDay(*t*))).
7. Set the [[Value]] property of the **`this`** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.37 Date.prototype.setUTCFullYear(year [, mon [, date]])

If *mon* is not specified, this behaves as if *mon* 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. Call ToNumber(*year*).
3. If *mon* is not specified, compute MonthFromTime(*t*); otherwise, call ToNumber(*mon*).
4. If *date* is not specified, compute DateFromTime(*t*); otherwise, call ToNumber(*date*).
5. Compute MakeDay(Result(2), Result(3), Result(4)).
6. Compute MakeDate(Result(5), TimeWithinDay(*t*)).
7. Set the [[Value]] property of the **`this`** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.38 Date.prototype.setYear(year)

NOTE This function is not part of this specification. The function **`setFullYear`** is much to be preferred for nearly all purposes, because it avoids the "year 2000 problem." If implemented, **setYear** may follow the following rules:

1. Let *t* be the result of LocalTime(this time value); but if this time value is **NaN**, let *t* be **+0**.
2. Call ToNumber(*year*).
3. If Result(2) is **NaN**, set the [[Value]] property of the **`this`** value to **NaN** and return **NaN**.
4. If Result(2) is not **NaN** and 0 = ToInteger(Result(2)) = 99 then Result(4) is ToInteger(Result(2)) + 1900. Otherwise, Result(4) is Result(2).
5. Compute MakeDay(Result(4), MonthFromTime(*t*), DateFromTime(*t*)).
6. Compute UTC(MakeDate(Result(5), TimeWithinDay(*t*))).
7. Set the [[Value]] property of the **`this`** value to TimeClip(Result(6)).
8. Return the value of the [[Value]] property of the **`this`** value.

15.9.5.39 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 a convenient, human-readable form appropriate to the geographic or cultural locale.

15.9.5.40 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.

15.9.5.41 Date.prototype.toGMTString()

The function object that is the initial value of **`Date.prototype.toGMTString`** is the same function object that is the initial value of **`Date.prototype.toUTCString`**. The **`toGMTString`** property is provided principally for compatibility with old code. It is recommended that the **`toUTCString`** property be used in new ECMAScript code.

15.9.6 Properties of Date Instances

Date instances have no special properties beyond those inherited from the Date prototype object.

16 Errors

This specification specifies the last possible moment an error occurs. A given implementation may generate errors sooner (e.g., at compile-time). Doing so may cause differences in behaviour among implementations. Notably, if runtime errors become catchable in future versions, a given error would not be catchable if an implementation generates the error at compile-time rather than runtime.

An ECMAScript compiler should detect errors at compile time in all code presented to it, even code that detailed analysis might prove to be “dead” (never executed). A programmer should not rely on the trick of placing code within an **`if (false)`** statement, for example, to try to suppress compile-time error detection.

In general, if a compiler can prove that a construct cannot execute without error under any circumstances, then it may issue a compile-time error even though the construct might never be executed at all.

Printed copies can be ordered from:

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

Fax:	+41 22 849.60.01
Internet:	documents@ecma.ch

Files can be downloaded from our FTP site, **ftp.ecma.ch**. This Standard is available from library **ECMA-ST** as a compacted, self-expanding file in MSWord 6.0 format (file E262-DOC.EXE) and as an Acrobat PDF file (file E262-PDF.PDF). File E262-EXP.TXT gives a short presentation of the Standard.

Our web site, http://www.ecma.ch, gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.