
Information technology — ECMAScript language specification

Technologies de l'information — Spécifications du langage ECMAScript



Contents

1 Scope	1
2 Conformance	1
3 Normative References	1
4 Overview	1
4.1 Web Scripting	2
4.2 Language Overview	2
4.2.1 Objects	2
4.3 Definitions	3
4.3.1 Type	3
4.3.2 Primitive value	3
4.3.3 Object	4
4.3.4 Constructor	4
4.3.5 Prototype	4
4.3.6 Native object	4
4.3.7 Built-in object	4
4.3.8 Host object	4
4.3.9 Undefined value	4
4.3.10 Undefined type	4
4.3.11 Null value	4
4.3.12 Null type	4
4.3.13 Boolean value	4
4.3.14 Boolean type	4
4.3.15 Boolean object	4
4.3.16 String value	5
4.3.17 String type	5
4.3.18 String object	5
4.3.19 Number value	5
4.3.20 Number type	5
4.3.21 Number object	5
4.3.22 Infinity	5
4.3.23 NaN	5
5 Notational Conventions	5
5.1 Syntactic and Lexical Grammars	5
5.1.1 Context-Free Grammars	5
5.1.2 The lexical grammar	5
5.1.3 The numeric string grammar	6
5.1.4 The syntactic grammar	6

© ISO/IEC 1998

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

5.1.5 Grammar Notation	6
5.2 Algorithm Conventions	8
6 Source Text	9
7 Lexical Conventions	9
7.1 White Space	9
7.2 Line Terminators	10
7.3 Comments	10
7.4 Tokens	11
7.4.1 Reserved Words	11
7.4.2 Keywords	11
7.4.3 Future Reserved Words	12
7.5 Identifiers	12
7.6 Punctuators	13
7.7 Literals	13
7.7.1 Null Literals	13
7.7.2 Boolean Literals	13
7.7.3 Numeric Literals	14
7.7.4 String Literals	16
7.8 Automatic semicolon insertion	19
7.8.1 Rules of automatic semicolon insertion	19
7.8.2 Examples of Automatic Semicolon Insertion	20
8 Types	21
8.1 The Undefined type	21
8.2 The Null type	21
8.3 The Boolean type	21
8.4 The String type	21
8.5 The Number type	21
8.6 The Object type	23
8.6.1 Property attributes	23
8.6.2 Internal Properties and Methods	23
8.7 The Reference Type	25
8.7.1 GetBase(V)	26
8.7.2 GetPropertyName(V)	26
8.7.3 GetValue(V)	26
8.7.4 PutValue(V, W)	26
8.8 The List type	26
8.9 The Completion Type	26
9 Type Conversion	27
9.1 ToPrimitive	27
9.2 ToBoolean	27
9.3 ToNumber	28
9.3.1 ToNumber Applied to the String Type	28
9.4 ToInteger	31
9.5 ToInt32: (signed 32 bit integer)	31
9.6 ToUint32: (unsigned 32 bit integer)	31
9.7 ToUint16: (unsigned 16 bit integer)	32
9.8 ToString	32
9.8.1 ToString Applied to the Number Type	32
9.9 ToObject	33
10 Execution Contexts	33
10.1 Definitions	33
10.1.1 Function Objects	33
10.1.2 Types of Executable Code	34

10.1.3 Variable instantiation	34
10.1.4 Scope Chain and Identifier Resolution	35
10.1.5 Global Object	35
10.1.6 Activation object	35
10.1.7 This	35
10.1.8 Arguments Object	36
10.2 Entering An Execution Context	36
10.2.1 Global Code	36
10.2.2 Eval Code	36
10.2.3 Function and Anonymous Code	36
10.2.4 Implementation-supplied Code	37
11 Expressions	37
11.1 Primary Expressions	37
11.1.1 The <code>this</code> keyword	37
11.1.2 Identifier reference	37
11.1.3 Literal reference	37
11.1.4 The Grouping Operator	37
11.2 Left-Hand-Side Expressions	38
11.2.1 Property Accessors	38
11.2.2 The <code>new</code> operator	39
11.2.3 Function Calls	39
11.2.4 Argument Lists	39
11.3 Postfix expressions	40
11.3.1 Postfix increment operator	40
11.3.2 Postfix decrement operator	40
11.4 Unary operators	41
11.4.1 The <code>delete</code> operator	41
11.4.2 The <code>void</code> operator	41
11.4.3 The <code>typeof</code> operator	41
11.4.4 Prefix increment operator	42
11.4.5 Prefix decrement operator	42
11.4.6 Unary <code>+</code> operator	42
11.4.7 Unary <code>-</code> operator	42
11.4.8 The bitwise NOT operator (<code>~</code>)	42
11.4.9 Logical NOT operator (<code>!</code>)	43
11.5 Multiplicative operators	43
11.5.1 Applying the <code>*</code> operator	43
11.5.2 Applying the <code>/</code> operator	43
11.5.3 Applying the <code>%</code> operator	44
11.6 Additive operators	45
11.6.1 The addition operator (<code>+</code>)	45
11.6.2 The subtraction operator (<code>-</code>)	45
11.6.3 Applying the additive operators (<code>+</code> , <code>-</code>) to numbers	45
11.7 Bitwise shift operators	46
11.7.1 The left shift operator (<code><<</code>)	46
11.7.2 The signed right shift operator (<code>>></code>)	46
11.7.3 The unsigned right shift operator (<code>>>></code>)	47
11.8 Relational operators	47
11.8.1 The less-than operator (<code><</code>)	47
11.8.2 The greater-than operator (<code>></code>)	47
11.8.3 The less-than-or-equal operator (<code><=</code>)	47
11.8.4 The greater-than-or-equal operator (<code>>=</code>)	48
11.8.5 The abstract relational comparison algorithm	48
11.9 Equality operators	48
11.9.1 The equals operator (<code>==</code>)	49
11.9.2 The does-not-equals operator (<code>!=</code>)	49

11.9.3 The abstract equality comparison algorithm	49
11.10 Binary bitwise operators	50
11.11 Binary logical operators	50
11.12 Conditional operator (?:)	51
11.13 Assignment operators	51
11.13.1 Simple Assignment (=)	52
11.13.2 Compound assignment (op=)	52
11.14 Comma operator (,)	52
12 Statements	52
12.1 Block	53
12.2 Variable statement	53
12.3 Empty statement	54
12.4 Expression statement	54
12.5 The if statement	55
12.6 Iteration statements	55
12.6.1 The while statement	55
12.6.2 The for statement	56
12.6.3 The for..in statement	57
12.7 The continue statement	57
12.8 The break statement	58
12.9 The return statement	58
12.10 The with statement	58
13 Function Definition	59
14 Program	59
15 Native ECMAScript objects	60
15.1 The Global Object	61
15.1.1 Value properties of the Global Object	61
15.1.2 Function properties of the Global Object	61
15.1.3 Constructor Properties of the Global Object	64
15.1.4 Other Properties of the Global Object	64
15.2 Object Objects	64
15.2.1 The Object Constructor Called as a Function	64
15.2.2 The Object Constructor	64
15.2.3 Properties of the Object Constructor	65
15.2.4 Properties of the Object Prototype Object	65
15.2.5 Properties of Object Instances	65
15.3 Function Objects	65
15.3.1 The Function Constructor Called as a Function	65
15.3.2 The Function Constructor	65
15.3.3 Properties of the Function Constructor	67
15.3.4 Properties of the Function Prototype Object	67
15.3.5 Properties of Function Instances	67
15.4 Array Objects	67
15.4.1 The Array Constructor Called as a Function	67
15.4.2 The Array Constructor	68
15.4.3 Properties of the Array Constructor	68
15.4.4 Properties of the Array Prototype Object	68
15.4.5 Properties of Array Instances	71
15.5 String Objects	71
15.5.1 The String Constructor Called as a Function	71
15.5.2 The String Constructor	71
15.5.3 Properties of the String Constructor	72
15.5.4 Properties of the String Prototype Object	72
15.5.5 Properties of String Instances	75

15.6 Boolean Objects	76
15.6.1 The Boolean Constructor Called as a Function	76
15.6.2 The Boolean Constructor	76
15.6.3 Properties of the Boolean Constructor	76
15.6.4 Properties of the Boolean Prototype Object	76
15.6.5 Properties of Boolean Instances	77
15.7 Number Objects	77
15.7.1 The Number Constructor Called as a Function	77
15.7.2 The Number Constructor	77
15.7.3 Properties of the Number Constructor	77
15.7.4 Properties of the Number Prototype Object	78
15.7.5 Properties of Number Instances	78
15.8 The Math Object	78
15.8.1 Value Properties of the Math Object	78
15.8.2 Function Properties of the Math Object	79
15.9 Date Objects	84
15.9.1 Overview of Date Objects and Definitions of Internal Operators	84
15.9.2 The Date Constructor Called As a Function	87
15.9.3 The Date Constructor	88
15.9.4 Properties of the Date Constructor	90
15.9.5 Properties of the Date Prototype Object	93
15.9.6 Properties of Date Instances	98
16 Errors	98

Foreword

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

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

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

Information technology - ECMAScript language specification

1 Scope

This Standard defines the ECMAScript scripting language.

2 Conformance

A conforming implementation of ECMAScript must provide and support all the types, values, objects, properties, functions, and program syntax 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 7.4.3.

3 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 9899:1990, *Programming languages – C*.

ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*.

ISO/IEC 646:1991, *Information technology – 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 clause contains a non-normative overview of the ECMAScript language.

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

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

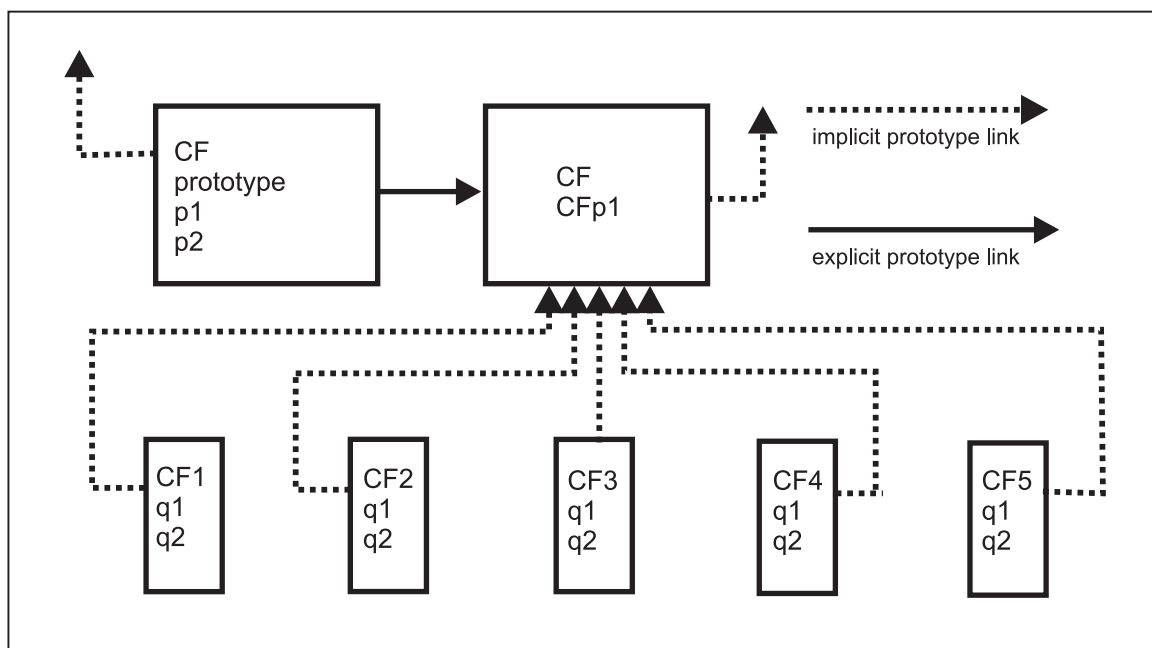
4.2.1 Objects

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 CFp1. The constructor, CF, has two properties itself, named p1 and p2, which are not visible to CFp1, CF1, CF2, CF3, CF4, or CF5. The property named CFp1 in CFp1 is shared by CF1, CF2, CF3, CF4, and CF5, as are any properties found in CFp1's implicit prototype chain which are not named q1, q2, or CFp1. Notice that there is no implicit prototype link between CFp1 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 CFp1.

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 clause 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 clause 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 7.8). 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 clause 9.3.1.

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

5.1.4 The syntactic grammar

The *syntactic grammar* for ECMAScript is given in clauses 11, 12, 13, and 14. This grammar has ECMAScript tokens defined by the lexical grammar as its terminal symbols (see 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 clauses 11, 12, 13, and 14 is actually not a complete account of which token sequences are accepted as correct ECMAScript programs. Certain additional token sequences are also accepted, namely, those that would be described by the grammar if only semicolons were added to the sequence in certain places (such as before line terminator characters). Furthermore, certain token sequences that are described by the grammar are not considered acceptable if a terminator character appears in certain “awkward” places.

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

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

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

The notation “x modulo y” (y must be finite and nonzero) computes a value k of the same sign as y such that abs(k) < abs(y) and x−k = q·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:

<i>Unicode Value</i>	<i>Name</i>	<i>Formal Name</i>
\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 7.8).

The following characters are considered to be line terminators:

<i>Unicode Value</i>	<i>Name</i>	<i>Formal Name</i>
\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 7.8.2).

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 *DecimalDigits*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* . *ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^e , where e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* . *DecimalDigits* *ExponentPart* is (the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times 10^{-n})) times 10^e , where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* :: . *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n} , where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalLiteral* :: . *DecimalDigits* *ExponentPart* is the MV of *DecimalDigits* times 10^{e-n} , where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* *ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^e , where e is the MV of *ExponentPart*.
- The MV of *DecimalIntegerLiteral* :: 0 is 0.
- The MV of *DecimalIntegerLiteral* :: *NonZeroDigit* *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 8.5), unless the literal is a *DecimalLiteral* and the literal has more than 20 significant digits, in which case the number value may be either the number value for the MV of a literal produced by replacing each significant digit after the 20th with a **0** digit or the number value for the MV of a literal produced by replacing each significant digit after the 20th with a **0** digit and then incrementing the literal at the 20th significant digit position. A digit is *significant* if it is not part of an *ExponentPart* and

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

7.7.4 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 7.7.3. *SourceCharacter* is described in clauses 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 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:

<i>Escape Sequence</i>	<i>Unicode Value</i>	<i>Name</i>	<i>Symbol</i>
\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* :: $\backslash \text{OctalDigit OctalDigit}$ 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* :: $\backslash \text{ZeroToThree OctalDigit OctalDigit}$ 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* :: $\backslash \text{u HexDigit HexDigit HexDigit HexDigit}$ 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 $\backslash$. 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 $\backslash \text{n}$ or $\backslash \text{u000A}$.

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:
 3. If the parser is not currently parsing the header of a **for** statement, a semicolon is automatically inserted before the restricted token.
 4. 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 clause 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 **$+\infty$** ; if -2^{1024} was chosen, replace it with **$-\infty$** ; if **+0** was chosen, replace it with **-0** if and only if x is less than zero; any other chosen value is used unchanged. The result is the number value for x . (This procedure corresponds exactly to the behaviour of the IEEE 754 “round to nearest” mode.)

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

8.6 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 (12.6.3)
DontDelete	Attempts to delete the property will be ignored. See the description of the delete operator in 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
<code>[[Prototype]]</code>	none	The prototype of this object.
<code>[[Class]]</code>	none	A string value indicating the kind of this object.
<code>[[Value]]</code>	none	Internal state information associated with this object.
<code>[[Get]]</code>	(PropertyName)	Returns the value of the property.
<code>[[Put]]</code>	(PropertyName, Value)	Sets the specified property to Value.
<code>[[CanPut]]</code>	(PropertyName)	Returns a boolean value indicating whether a <code>[[Put]]</code> operation with the specified PropertyName will succeed.
<code>[[HasProperty]]</code>	(PropertyName)	Returns a boolean value indicating whether the object already has a member with the given name.
<code>[[Delete]]</code>	(PropertyName)	Removes the specified property from the object.
<code>[[DefaultValue]]</code>	(Hint)	Returns a default value for the object, which should be a primitive value (not an object or reference).
<code>[[Construct]]</code>	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 <i>constructors</i> .
<code>[[Call]]</code>	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 <i>functions</i> .

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 8.6.2.1, 8.6.2.2, 8.6.2.3, 8.6.2.4, and 8.6.2.5, respectively, except that Array objects have a slightly different implementation of the `[[Put]]` method (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 (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)`

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

1. If *O* has a property with name *P*, return **true**.
2. If the `[[Prototype]]` of *O* is **null**, return **false**.
3. Call the `[[HasProperty]]` method of `[[Prototype]]` with property name *P*.
4. 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 15.9), 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*.
- **GetProperty(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*.