

Information technology — ECMAScript language specification

ICS 35.060

National foreword

This British Standard reproduces verbatim ISO/IEC 16262:2002 and implements it as the UK national standard.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and system software interfaces, which has the responsibility to:

- aid enquirers to understand the text;
- present to the responsible international/European committee any enquiries on the interpretation, or proposals for change, and keep the UK interests informed;
- monitor related international and European developments and promulgate them in the UK.

A list of organizations represented on this committee can be obtained on request to its secretary.

Cross-references

The British Standards which implement international or European publications referred to in this document may be found in the BSI Standards Catalogue under the section entitled “International Standards Correspondence Index”, or by using the “Find” facility of the BSI Standards Electronic Catalogue.

A British Standard does not purport to include all the necessary provisions of a contract. Users of British Standards are responsible for their correct application.

Compliance with a British Standard does not of itself confer immunity from legal obligations.

Summary of pages

This document comprises a front cover, an inside front cover, the ISO/IEC title page, pages ii to ix, a blank page, pages 1 to 169 and a back cover.

The BSI copyright date displayed in this document indicates when the document was last issued.

This British Standard, having been prepared under the direction of the DISC Board, was published under the authority of the Standards Policy and Strategy Committee on 2 July 2002

© BSI 2 July 2002

Amendments issued since publication

Amd. No.	Date	Comments

ISBN 0 580 39939 7

**Information technology — ECMAScript
language specification**

Technologies de l'information — ECMAScript spécifications du langage



Reference number
ISO/IEC 16262:2002(E)

Contents	Page
1 Scope	1
2 Conformance.....	1
3 Normative References.....	1
4 Overview	2
4.1 Web Scripting.....	2
4.2 Language Overview	2
4.2.1 Objects	3
4.3 Terms and Definitions	4
4.3.1 Type.....	4
4.3.2 Primitive Value.....	4
4.3.3 Object	4
4.3.4 Constructor	4
4.3.5 Prototype	5
4.3.6 Native Object	5
4.3.7 Built-in Object	5
4.3.8 Host Object.....	5
4.3.9 Undefined Value	5
4.3.10 Undefined Type	5
4.3.11 Null Value	5
4.3.12 Null Type	5
4.3.13 Boolean Value	5
4.3.14 Boolean Type	5
4.3.15 Boolean Object.....	5
4.3.16 String Value	6
4.3.17 String Type	6
4.3.18 String Object.....	6
4.3.19 Number Value.....	6
4.3.20 Number Type.....	6
4.3.21 Number Object	6
4.3.22 Infinity	6
4.3.23 NaN	6
5 Notational Conventions	6
5.1 Syntactic and Lexical Grammars	6
5.1.1 Context-Free Grammars	6
5.1.2 The Lexical and RegExp Grammars	7
5.1.3 The Numeric String Grammar	7
5.1.4 The Syntactic Grammar.....	7
5.1.5 Grammar Notation	8
5.2 Algorithm Conventions.....	10
6 Source Text	11
7 Lexical Conventions.....	11
7.1 Unicode Format-Control Characters	12
7.2 White Space	12
7.3 Line Terminators	12
7.4 Comments	13
7.5 Tokens.....	14

7.5.1 Reserved Words	14
7.5.2 Keywords	14
7.5.3 Future Reserved Words	14
7.6 Identifiers	15
7.7 Punctuators	16
7.8 Literals	16
7.8.1 Null Literals	16
7.8.2 Boolean Literals	17
7.8.3 Numeric Literals	17
7.8.4 String Literals	19
7.8.5 Regular Expression Literals	21
7.9 Automatic Semicolon Insertion	22
7.9.1 Rules of Automatic Semicolon Insertion	22
7.9.2 Examples of Automatic Semicolon Insertion	23
8 Types	24
8.1 The Undefined Type	24
8.2 The Null Type	24
8.3 The Boolean Type	24
8.4 The String Type	24
8.5 The Number Type	25
8.6 The Object Type	26
8.6.1 Property Attributes	26
8.6.2 Internal Properties and Methods	26
8.7 The Reference Type	29
8.7.1 GetValue (V)	30
8.7.2 PutValue (V, W)	30
8.8 The List Type	30
8.9 The Completion Type	30
9 Type Conversion	30
9.1 ToPrimitive	30
9.2 ToBoolean	31
9.3 ToNumber	31
9.3.1 ToNumber Applied to the String Type	31
9.4 ToInteger	34
9.5 ToInt32: (Signed 32 Bit Integer)	34
9.6 ToUint32: (Unsigned 32 Bit Integer)	35
9.7 ToUint16: (Unsigned 16 Bit Integer)	35
9.8 ToString	35
9.8.1 ToString Applied to the Number Type	36
9.9 ToObject	37
10 Execution Contexts	37
10.1 Definitions	37
10.1.1 Function Objects	37
10.1.2 Types of Executable Code	37
10.1.3 Variable Instantiation	38
10.1.4 Scope Chain and Identifier Resolution	38
10.1.5 Global Object	38
10.1.6 Activation Object	39
10.1.7 This	39
10.1.8 Arguments Object	39
10.2 Entering An Execution Context	39
10.2.1 Global Code	39
10.2.2 Eval Code	40
10.2.3 Function Code	40
11 Expressions	40

11.1 Primary Expressions	40
11.1.1 The <code>this</code> Keyword.....	40
11.1.2 Identifier Reference	40
11.1.3 Literal Reference	40
11.1.4 Array Initialiser	40
11.1.5 Object Initialiser	42
11.1.6 The Grouping Operator	43
11.2 Left-Hand-Side Expressions	43
11.2.1 Property Accessors	44
11.2.2 The <code>new</code> Operator	44
11.2.3 Function Calls	45
11.2.4 Argument Lists.....	45
11.2.5 Function Expressions	45
11.3 Postfix Expressions.....	46
11.3.1 Postfix Increment Operator	46
11.3.2 Postfix Decrement Operator	46
11.4 Unary Operators.....	46
11.4.1 The <code>delete</code> Operator	46
11.4.2 The <code>void</code> Operator.....	47
11.4.3 The <code>typeof</code> Operator	47
11.4.4 Prefix Increment Operator	47
11.4.5 Prefix Decrement Operator	47
11.4.6 Unary <code>+</code> Operator	47
11.4.7 Unary <code>-</code> Operator	48
11.4.8 Bitwise NOT Operator (<code>~</code>).....	48
11.4.9 Logical NOT Operator (<code>!</code>).....	48
11.5 Multiplicative Operators.....	48
11.5.1 Applying the <code>*</code> Operator	49
11.5.2 Applying the <code>/</code> Operator	49
11.5.3 Applying the <code>%</code> Operator	49
11.6 Additive Operators	50
11.6.1 The Addition operator (<code>+</code>)	50
11.6.2 The Subtraction Operator (<code>-</code>)	51
11.6.3 Applying the Additive Operators (<code>+</code> , <code>-</code>) to Numbers	51
11.7 Bitwise Shift Operators.....	51
11.7.1 The Left Shift Operator (<code><<</code>)	51
11.7.2 The Signed Right Shift Operator (<code>>></code>)	52
11.7.3 The Unsigned Right Shift Operator (<code>>>></code>)	52
11.8 Relational Operators	52
11.8.1 The Less-than Operator (<code><</code>).....	53
11.8.2 The Greater-than Operator (<code>></code>)	53
11.8.3 The Less-than-or-equal Operator (<code><=</code>)	53
11.8.4 The Greater-than-or-equal Operator (<code>>=</code>)	53
11.8.5 The Abstract Relational Comparison Algorithm	54
11.8.6 The <code>instanceof</code> operator	54
11.8.7 The <code>in</code> operator	54
11.9 Equality Operators	55
11.9.1 The Equals Operator (<code>==</code>)	55
11.9.2 The Does-not-equals Operator (<code>!=</code>).....	55
11.9.3 The Abstract Equality Comparison Algorithm	56
11.9.4 The Strict Equals Operator (<code>===</code>)	57
11.9.5 The Strict Does-not-equal Operator (<code>!==</code>)	57
11.9.6 The Strict Equality Comparison Algorithm.....	57
11.10 Binary Bitwise Operators	57
11.11 Binary Logical Operators	58
11.12 Conditional Operator (<code>?:</code>)	59
11.13 Assignment Operators	60
11.13.1 Simple Assignment (<code>=</code>)	60

11.13.2 Compound Assignment (<code>op=</code>)	60
11.14 Comma Operator (<code>,</code>)	60
12 Statements.....	61
12.1 Block.....	61
12.2 Variable statement	62
12.3 Empty Statement.....	63
12.4 Expression Statement	63
12.5 The <code>if</code> Statement	64
12.6 Iteration Statements	64
12.6.1 The <code>do-while</code> Statement.....	65
12.6.2 The <code>while</code> statement.....	65
12.6.3 The <code>for</code> Statement.....	65
12.6.4 The <code>for-in</code> Statement.....	66
12.7 The <code>continue</code> Statement	67
12.8 The <code>break</code> Statement.....	67
12.9 The <code>return</code> Statement.....	68
12.10 The <code>with</code> Statement	68
12.11 The <code>switch</code> Statement.....	68
12.12 Labelled Statements	70
12.13 The <code>throw</code> statement	70
12.14 The <code>try</code> statement.....	71
13 Function Definition	72
13.1 Definitions	73
13.1.1 Equated Grammar Productions.....	73
13.1.2 Joined Objects.....	73
13.2 Creating Function Objects.....	73
13.2.1 <code>[[Call]]</code>	75
13.2.2 <code>[[Construct]]</code>	75
14 Program	75
15 Native ECMAScript Objects	76
15.1 The Global Object	77
15.1.1 Value Properties of the Global Object	77
15.1.2 Function Properties of the Global Object	77
15.1.3 URI Handling Function Properties.....	79
15.1.4 Constructor Properties of the Global Object.....	83
15.1.5 Other Properties of the Global Object	84
15.2 Object Objects.....	84
15.2.1 The Object Constructor Called as a Function	84
15.2.2 The Object Constructor	84
15.2.3 Properties of the Object Constructor	84
15.2.4 Properties of the Object Prototype Object	85
15.2.5 Properties of Object Instances	86
15.3 Function Objects	86
15.3.1 The Function Constructor Called as a Function	86
15.3.2 The Function Constructor	86
15.3.3 Properties of the Function Constructor.....	87
15.3.4 Properties of the Function Prototype Object	87
15.3.5 Properties of Function Instances	88
15.4 Array Objects.....	88
15.4.1 The Array Constructor Called as a Function	89
15.4.2 The Array Constructor	89
15.4.3 Properties of the Array Constructor.....	89
15.4.4 Properties of the Array Prototype Object.....	90
15.4.5 Properties of Array Instances	97

15.5 String Objects.....	98
15.5.1 The String Constructor Called as a Function	98
15.5.2 The String Constructor	98
15.5.3 Properties of the String Constructor	99
15.5.4 Properties of the String Prototype Object.....	99
15.5.5 Properties of String Instances	106
15.6 Boolean Objects.....	107
15.6.1 The Boolean Constructor Called as a Function	107
15.6.2 The Boolean Constructor	107
15.6.3 Properties of the Boolean Constructor	107
15.6.4 Properties of the Boolean Prototype Object.....	107
15.6.5 Properties of Boolean Instances	108
15.7 Number Objects	108
15.7.1 The Number Constructor Called as a Function.....	108
15.7.2 The Number Constructor.....	108
15.7.3 Properties of the Number Constructor.....	108
15.7.4 Properties of the Number Prototype Object	109
15.7.5 Properties of Number Instances.....	112
15.8 The Math Object.....	112
15.8.1 Value Properties of the Math Object	112
15.8.2 Function Properties of the Math Object.....	113
15.9 Date Objects	118
15.9.1 Overview of Date Objects and Definitions of Internal Operators	118
15.9.2 The Date Constructor Called as a Function	122
15.9.3 The Date Constructor	122
15.9.4 Properties of the Date Constructor.....	123
15.9.5 Properties of the Date Prototype Object.....	124
15.9.6 Properties of Date Instances	131
15.10 RegExp (Regular Expression) Objects	131
15.10.1 Patterns	131
15.10.2 Pattern Semantics	133
15.10.3 The RegExp Constructor Called as a Function.....	145
15.10.4 The RegExp Constructor.....	145
15.10.5 Properties of the RegExp Constructor.....	146
15.10.6 Properties of the RegExp Prototype Object	146
15.10.7 Properties of RegExp Instances.....	147
15.11 Error Objects	148
15.11.1 The Error Constructor Called as a Function.....	148
15.11.2 The Error Constructor.....	148
15.11.3 Properties of the Error Constructor	148
15.11.4 Properties of the Error Prototype Object	148
15.11.5 Properties of Error Instances.....	149
15.11.6 Native Error Types Used in This Standard.....	149
15.11.7 <i>NativeError</i> Object Structure	149
16 Errors	151
Annex A (informative) Grammar Summary.....	152
A.1 Lexical Grammar	152
A.2 Number Conversions	156
A.3 Expressions	157
A.4 Statements.....	161
A.5 Functions and Programs.....	163
A.6 Universal Resource Identifier Character Classes.....	163
A.7 Regular Expressions.....	164
Annex B (informative) Compatibility	166
B.1 Additional Syntax	166
B.1.1 Numeric Literals	166

B.1.2 String Literals 166

B.2 Additional Properties..... 167

 B.2.1 escape (string)..... 167

 B.2.2 unescape (string) 168

 B.2.3 String.prototype.substr (start, length) 168

 B.2.4 Date.prototype.getYear ()..... 169

 B.2.5 Date.prototype.setYear (year)..... 169

 B.2.6 Date.prototype.toGMTString ()..... 169

Foreword

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

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 3.

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

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

ISO/IEC 16262 was prepared by ECMA (as ECMA-262) and was adopted, under a special “fast-track procedure”, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*, in parallel with its approval by national bodies of ISO and IEC.

This second edition cancels and replaces the first edition (ISO/IEC 16262:1998), which has been technically revised.

Annexes A and B of this International Standard are for information only.

Information technology — ECMAScript language specification

1 Scope

This International Standard defines the ECMAScript scripting language.

2 Conformance

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

A conforming implementation of this International Standard shall interpret characters in conformance with the Unicode Standard, Version 2.1 or later, and ISO/IEC 10646-1 with either UCS-2 or UTF-16 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. If the adopted encoding form is not otherwise specified, it is presumed to be the UTF-16 encoding form.

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

A conforming implementation of ECMAScript is permitted to support program and regular expression syntax not described in this specification. In particular, a conforming implementation of ECMAScript is permitted to support program syntax that makes use of the “future reserved words” listed in section 7.5.3 of this specification.

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this International Standard. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply. However, parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the normative documents indicated below. For undated references, the latest edition of the normative document referred to applies. Members of ISO and IEC maintain registers of currently valid International Standards.

ISO/IEC 9899:1999 *Programming languages – C*, including Technical Corrigendum 1

ISO/IEC 10646-1:2000 *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*, including Amendment 1

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

Unicode Inc. (1998), *Unicode Technical Report #8: The Unicode Standard™*, Version 2.1.

Unicode Inc. (1998), *Unicode Technical Report #15: Unicode Normalization Forms*.

ANSI/IEEE Std 754-1985: *IEEE Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronic 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. To accommodate non-professional programmers, some aspects of the language may be somewhat less strict.

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 zero or more **attributes** that 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** that round out the definition of ECMAScript entities. These built-in objects include the **Global** object, the **Object** object, the **Function** object, the **Array** object, the **String** object, the **Boolean** object, the **Number** object, the **Math** object, the **Date** object, the **RegExp** object and the Error objects **Error**, **EvalError**, **RangeError**, **ReferenceError**, **SyntaxError**, **TypeError** and **URIError**.

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.

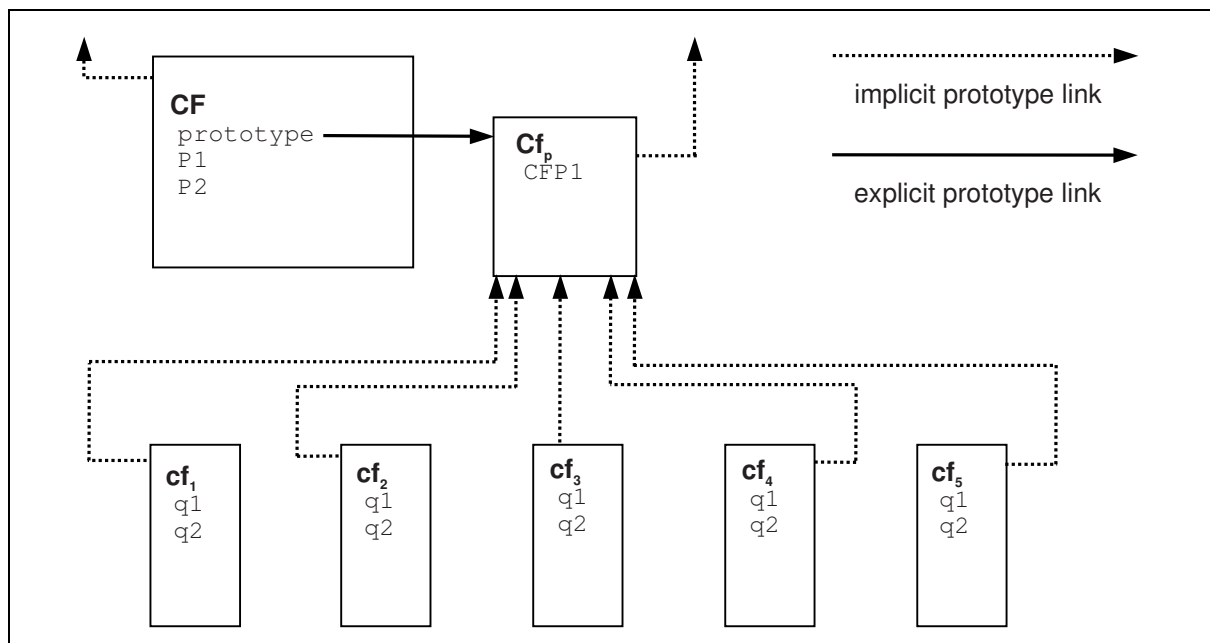
4.2.1 Objects

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 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` (but not by CF), as are any properties found in `CFp`'s implicit prototype chain that are not named `q1`, `q2`, or `CFP1`. Notice that there is no implicit prototype link between CF and `CFp`.

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 Terms and Definitions

The following are informal terms and 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 an 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.

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 16-bit unsigned integer values.

NOTE Although each value usually represents a single 16-bit unit of UTF-16 text, the language does not place any restrictions or requirements on the values except that they be 16-bit unsigned integers.

4.3.17 String Type

The type **String** is the set of all string values.

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 by calling the String constructor as a function (section 15.5.1).

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 represents 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 by calling the Number constructor as a function (section 15.7.1).

4.3.22 Infinity

The primitive value **Infinity** represents the positive infinite number value. This value is a member of the Number type.

4.3.23 NaN

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

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 zero or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified alphabet.

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

5.1.2 The Lexical and RegExp Grammars

A *lexical grammar* for ECMAScript is given in 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 *InputElementDiv* or *InputElementRegExp*, 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 (section 7.9). Simple white space and single-line comments are discarded and do not appear in the stream of input elements for the syntactic grammar. A *MultiLineComment* (that is, a comment of the form “/*...*/” regardless of whether it spans more than one line) is likewise simply discarded if it contains no line terminator; but if a *MultiLineComment* contains one or more line terminators, then it is replaced by a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.

A *RegExp grammar* for ECMAScript is given in section 15.10. This grammar also has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol *Pattern*, that describe how sequences of Unicode characters are translated into regular expression patterns.

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

5.1.3 The Numeric String Grammar

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 sections 11, 12, 13 and 14. This grammar has ECMAScript tokens defined by the lexical grammar as its terminal symbols (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 sections 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. All nonterminal characters specified in this way are to be understood as the appropriate Unicode character from the ASCII range, as opposed to any similar-looking characters from other Unicode ranges.

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

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 Initialiser*_{opt}

is a convenient abbreviation for:

VariableDeclaration :
Identifier
Identifier Initialiser

and that:

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

is a convenient abbreviation for:

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

which in turn is an abbreviation for:

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

which in turn is an abbreviation for:

```

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

```

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

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

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

```

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

```

```

DecimalDigits ::
    DecimalDigit
    DecimalDigits DecimalDigit

```

the definition

```

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

```

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

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

```

ReturnStatement :
    return [no LineTerminator here] Expressionopt ;

```

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:

```

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

```

which is merely a convenient abbreviation for:

NonZeroDigit :: **one of**

1
2
3
4
5
6
7
8
9

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

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

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

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

Finally, a few nonterminal symbols are described by a descriptive phrase in 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* (or zero) such that $\text{abs}(k) < \text{abs}(y)$ and $x - k = q \times y$ for some integer *q*.

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

NOTE $\text{floor}(x) = x - (x \text{ modulo } 1)$.

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

6 Source Text

ECMAScript source text is represented as a sequence of characters in the Unicode character encoding, version 2.1 or later, using the UTF-16 transformation format. The text is expected to have been normalised to Unicode Normalised Form C (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves.

SourceCharacter ::
any Unicode character

ECMAScript source text can contain any of the Unicode characters. All Unicode white space characters are treated as white space, and all Unicode line/paragraph separators are treated as line separators. Non-Latin Unicode characters are allowed in identifiers, string literals, regular expression literals and comments.

Throughout the rest of this document, the phrase “code point” and the word “character” will be used to refer to a 16-bit unsigned value used to represent a single 16-bit unit of UTF-16 text. The phrase “Unicode character” will be used to refer to the abstract linguistic or typographical unit represented by a single Unicode scalar value (which may be longer than 16 bits and thus may be represented by more than one code point). This only refers to entities represented by single Unicode scalar values: the components of a combining character sequence are still individual “Unicode characters,” even though a user might think of the whole sequence as a single character.

In string literals, regular expression literals and identifiers, any character (code point) may also be expressed as a Unicode escape sequence consisting of six characters, namely `\u` plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within a string literal or regular expression literal, the Unicode escape sequence contributes one character to the value of the literal. Within an identifier, the escape sequence contributes one character to the identifier.

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

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.

There are two goal symbols for the lexical grammar. The *InputElementDiv* symbol is used in those syntactic grammar contexts where a division (`/`) or division-assignment (`/=`) operator is permitted. The *InputElementRegExp* symbol is used in other syntactic grammar contexts.

Note that contexts exist in the syntactic grammar where both a division and a *RegularExpressionLiteral* are permitted by the syntactic grammar; however, since the lexical grammar uses the *InputElementDiv* goal symbol in such cases, the opening slash is not recognised as starting a regular expression literal in such a context. As a workaround, one may enclose the regular expression literal in parentheses.

Syntax

InputElementDiv ::
WhiteSpace
LineTerminator
Comment
Token
DivPunctuator

InputElementRegExp ::
WhiteSpace
LineTerminator
Comment
Token
RegularExpressionLiteral

7.1 Unicode Format-Control Characters

The Unicode format-control characters (i.e., the characters in category “Cf” in the Unicode Character Database such as LEFT-TO-RIGHT MARK or RIGHT-TO-LEFT MARK) are control codes used to control the formatting of a range of text in the absence of higher-level protocols for this (such as mark-up languages). It is useful to allow these in source text to facilitate editing and display.

The format control characters can occur anywhere in the source text of an ECMAScript program. These characters are removed from the source text before applying the lexical grammar. Since these characters are removed before processing string and regular expression literals, one must use a Unicode escape sequence (see section 7.6) to include a Unicode format-control character inside a string or regular expression literal.

7.2 White Space

White space characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other, but are otherwise insignificant. White space 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>Code Point Value</i>	<i>Name</i>	<i>Formal Name</i>
\u0009	Tab	<TAB>
\u000B	Vertical Tab	<VT>
\u000C	Form Feed	<FF>
\u0020	Space	<SP>
\u00A0	No-break space	<NBSP>
Other category “Zs”	Any other Unicode “space separator”	<USP>

Syntax

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

7.3 Line Terminators

Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any

two tokens, but there are a few places where they are forbidden by the syntactic grammar. A line terminator cannot occur within any token, not even a string. Line terminators also affect the process of automatic semicolon insertion (section 7.9).

The following characters are considered to be line terminators:

<i>Code Point Value</i>	<i>Name</i>	<i>Formal Name</i>
\u000A	Line Feed	<LF>
\u000D	Carriage Return	<CR>
\u2028	Line separator	<LS>
\u2029	Paragraph separator	<PS>

Syntax

LineTerminator ::

<LF>
<CR>
<LS>
<PS>

7.4 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 (section 7.9).

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

Syntax

Comment ::

MultiLineComment
SingleLineComment

MultiLineComment ::

/ MultiLineCommentChars_{opt} */*

MultiLineCommentChars ::

MultiLineNotAsteriskChar MultiLineCommentChars_{opt}
** PostAsteriskCommentChars_{opt}*

PostAsteriskCommentChars ::

MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars_{opt}
** PostAsteriskCommentChars_{opt}*

MultiLineNotAsteriskChar ::

SourceCharacter **but not** *asterisk* ***

MultiLineNotForwardSlashOrAsteriskChar ::

SourceCharacter **but not** *forward-slash* */* **or** *asterisk* ***

SingleLineComment ::
*// SingleLineCommentChars*_{opt}

SingleLineCommentChars ::
*SingleLineCommentChar SingleLineCommentChars*_{opt}

SingleLineCommentChar ::
SourceCharacter **but not** *LineTerminator*

7.5 Tokens

Syntax

Token ::
ReservedWord
Identifier
Punctuator
NumericLiteral
StringLiteral

7.5.1 Reserved Words

Description

Reserved words cannot be used as identifiers.

Syntax

ReservedWord ::
Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

7.5.2 Keywords

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

Syntax

Keyword :: **one of**

break	else	new	var
case	finally	return	void
catch	for	switch	while
continue	function	this	with
default	if	throw	
delete	in	try	
do	instanceof	typeof	

7.5.3 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	enum	int	short
boolean	export	interface	static
byte	extends	long	super
char	final	native	synchronized
class	float	package	throws
const	goto	private	transient
debugger	implements	protected	volatile
double	import	public	

7.6 Identifiers

Description

Identifiers are interpreted according to the grammar given in Section 5.16 of the upcoming version 3.0 of the Unicode standard, with some small modifications. This grammar is based on both normative and informative character categories specified by the Unicode standard. The characters in the specified categories in version 2.1 of the Unicode standard must be treated as in those categories by all conforming ECMAScript implementations; however, conforming ECMAScript implementations may allow additional legal identifier characters based on the category assignment from later versions of Unicode.

This International Standard specifies one departure from the grammar given in the Unicode standard: The dollar sign (\$) and the underscore (_) are permitted anywhere in an identifier. The dollar sign is intended for use only in mechanically generated code.

Unicode escape sequences are also permitted in identifiers, where they contribute a single character to the identifier, as computed by the CV of the *UnicodeEscapeSequence* (see section 7.8.4). The \ preceding the *UnicodeEscapeSequence* does not contribute a character to the identifier. A *UnicodeEscapeSequence* cannot be used to put a character into an identifier that would otherwise be illegal. In other words, if a \ *UnicodeEscapeSequence* sequence were replaced by its *UnicodeEscapeSequence*'s CV, the result must still be a valid *Identifier* that has the exact same sequence of characters as the original *Identifier*.

Two identifiers that are canonically equivalent according to the Unicode standard are *not* equal unless they are represented by the exact same sequence of code points (in other words, conforming ECMAScript implementations are only required to do bitwise comparison on identifiers). The intent is that the incoming source text has been converted to normalised form C before it reaches the compiler.

Syntax

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

IdentifierName ::
IdentifierStart
IdentifierName *IdentifierPart*

IdentifierStart ::
UnicodeLetter
 \$
 \ *UnicodeEscapeSequence*

IdentifierPart ::
IdentifierStart
UnicodeCombiningMark
UnicodeDigit
UnicodeConnectorPunctuation
 \ *UnicodeEscapeSequence*

UnicodeLetter
any character in the Unicode categories “Uppercase letter (Lu)”, “Lowercase letter (Ll)”, “Titlecase letter (Lt)”, “Modifier letter (Lm)”, “Other letter (Lo)”, or “Letter number (Nl)”.

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

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

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

UnicodeEscapeSequence
see section 7.8.4.

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

7.7 Punctuators

Syntax

Punctuator :: one of

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

DivPunctuator :: one of
/ /=

7.8 Literals

Syntax

Literal ::
NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral

7.8.1 Null Literals

Syntax

NullLiteral ::
null

Semantics

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

7.8.2 Boolean Literals

Syntax

BooleanLiteral ::
 true
 false

Semantics

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

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

7.8.3 Numeric Literals

Syntax

NumericLiteral ::
 DecimalLiteral
 HexIntegerLiteral

DecimalLiteral ::
 DecimalIntegerLiteral . *DecimalDigits*_{opt} *ExponentPart*_{opt}
 . *DecimalDigits* *ExponentPart*_{opt}
 DecimalIntegerLiteral *ExponentPart*_{opt}

DecimalIntegerLiteral ::
 0
 NonZeroDigit *DecimalDigits*_{opt}

DecimalDigits ::
 DecimalDigit
 DecimalDigits *DecimalDigit*

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

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

ExponentPart ::
 ExponentIndicator *SignedInteger*

ExponentIndicator :: **one of**
 e E

SignedInteger ::
 DecimalDigits
 + *DecimalDigits*
 - *DecimalDigits*

HexIntegerLiteral ::
 0x *HexDigit*
 0X *HexDigit*
 HexIntegerLiteral *HexDigit*

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

NOTE - For example:

3in

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

Semantics

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

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

- The MV of *HexDigit* :: ƒ or of *HexDigit* :: Ƒ is 15.
- The MV of *HexIntegerLiteral* :: 0x *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* :: 0x *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* :: *HexIntegerLiteral* *HexDigit* is (the MV of *HexIntegerLiteral* times 16) plus the MV of *HexDigit*.

Once the exact MV for a numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is **+0**; otherwise, the rounded value must be *the* number value for the MV (in the sense defined in section 8.5), unless the literal is a *DecimalLiteral* and the literal has more than 20 significant digits, in which case the number value may be either the number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th significant digit position. A digit is *significant* if it is not part of an *ExponentPart* and

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

7.8.4 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
0 [lookahead ∉ *DecimalDigit*]
HexEscapeSequence
UnicodeEscapeSequence

CharacterEscapeSequence ::

SingleEscapeCharacter
NonEscapeCharacter

SingleEscapeCharacter :: **one of**

' " \ b f n r t v

NonEscapeCharacter ::

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

```
EscapeCharacter ::
    SingleEscapeCharacter
    DecimalDigit
    x
    u
```

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

```
UnicodeEscapeSequence ::
    u HexDigit HexDigit HexDigit HexDigit
```

The definitions of the nonterminal *HexDigit* is given in section 7.8.3. *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 7.8.3.

- The SV of *StringLiteral* :: "" is the empty character sequence.
- The SV of *StringLiteral* :: ' ' is the empty character sequence.
- The SV of *StringLiteral* :: " *DoubleStringCharacters* " is the SV of *DoubleStringCharacters*.
- The SV of *StringLiteral* :: ' *SingleStringCharacters* ' is the SV of *SingleStringCharacters*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter* is a sequence of one character, the CV of *DoubleStringCharacter*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter DoubleStringCharacters* is a sequence of the CV of *DoubleStringCharacter* followed by all the characters in the SV of *DoubleStringCharacters* in order.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter* is a sequence of one character, the CV of *SingleStringCharacter*.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter SingleStringCharacters* is a sequence of the CV of *SingleStringCharacter* followed by all the characters in the SV of *SingleStringCharacters* in order.
- The 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* :: 0 [lookahead ≠ *DecimalDigit*] is a <NUL> character (Unicode value 0000).
- The CV of *EscapeSequence* :: *HexEscapeSequence* is the CV of the *HexEscapeSequence*.
- The CV of *EscapeSequence* :: *UnicodeEscapeSequence* is the CV of the *UnicodeEscapeSequence*.
- The CV of *CharacterEscapeSequence* :: *SingleEscapeCharacter* is the character whose code point value is determined by the *SingleEscapeCharacter* according to the following table:

Escape Sequence	Code Point Value	Name	Symbol
\b	\u0008	backspace	<BS>
\t	\u0009	horizontal tab	<HT>
\n	\u000A	line feed (new line)	<LF>
\v	\u000B	vertical tab	<VT>
\f	\u000C	form feed	<FF>
\r	\u000D	carriage return	<CR>
\"	\u0022	double quote	"
\'	\u0027	single quote	'
\\	\u005C	backslash	\

- The CV of *CharacterEscapeSequence* :: *NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.
- The CV of *NonEscapeCharacter* :: *SourceCharacter* **but not** *EscapeCharacter* **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *HexEscapeSequence* :: *x* *HexDigit* *HexDigit* is the character whose code point value is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.
- The CV of *UnicodeEscapeSequence* :: *u* *HexDigit* *HexDigit* *HexDigit* *HexDigit* is the character whose code point value 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`.

7.8.5 Regular Expression Literals

A regular expression literal is an input element that is converted to a `RegExp` object (section 15.10) when it is scanned. The object is created before evaluation of the containing program or function begins. Evaluation of the literal produces a reference to that object; it does not create a new object. Two regular expression literals in a program evaluate to regular expression objects that never compare as `===` to each other even if the two literals' contents are identical. A `RegExp` object may also be created at runtime by `new RegExp` (section 15.10.4) or calling the `RegExp` constructor as a function (section 15.10.3).

The productions below describe the syntax for a regular expression literal and are used by the input element scanner to find the end of the regular expression literal. The strings of characters comprising the *RegularExpressionBody* and the *RegularExpressionFlags* are passed uninterpreted to the regular expression constructor, which interprets them according to its own, more stringent grammar. An implementation may extend the regular expression constructor's grammar, but it should not extend the *RegularExpressionBody* and *RegularExpressionFlags* productions or the productions used by these productions.

Syntax

RegularExpressionLiteral ::
/ *RegularExpressionBody* / *RegularExpressionFlags*

RegularExpressionBody ::
RegularExpressionFirstChar *RegularExpressionChars*

RegularExpressionChars ::
[empty]
RegularExpressionChars *RegularExpressionChar*

RegularExpressionFirstChar ::
NonTerminator **but not** `*` **or** `\` **or** `/`
BackslashSequence

RegularExpressionChar ::
NonTerminator **but not** `\` **or** `/`
BackslashSequence

BackslashSequence ::
`\` *NonTerminator*

NonTerminator ::
SourceCharacter **but not** *LineTerminator*

RegularExpressionFlags ::
[empty]
RegularExpressionFlags *IdentifierPart*

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

Semantics

A regular expression literal stands for a value of the Object type. This value is determined in two steps: first, the characters comprising the regular expression's *RegularExpressionBody* and *RegularExpressionFlags* production expansions are collected uninterpreted into two strings *Pattern* and *Flags*, respectively. Then the **new RegExp** constructor is called with two arguments *Pattern* and *Flags* and the result becomes the value of the *RegularExpressionLiteral*. If the call to **new RegExp** generates an error, an implementation may, at its discretion, either report the error immediately while scanning the program, or it may defer the error until the regular expression literal is evaluated in the course of program execution.

7.9 Automatic Semicolon Insertion

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

7.9.1 Rules of Automatic Semicolon Insertion

- When, as the program is parsed from left to right, a token (called the *offending token*) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:
 - The offending token is separated from the previous token by at least one *LineTerminator*.
 - The offending token is `}`.
- 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.
- When, as the program is parsed from left to right, a token is encountered that is allowed by some production of the grammar, but the production is a *restricted production* and the token would be the first token for a terminal or nonterminal immediately following the annotation “[no *LineTerminator* here]” within the restricted production (and therefore such a token is called a restricted token), and the restricted token is separated from the previous token by at least one *LineTerminator*, then a semicolon is automatically inserted before the restricted token.

However, there is an additional overriding condition on the preceding rules: a semicolon is never inserted automatically if the semicolon would then be parsed as an empty statement or if that semicolon would become one of the two semicolons in the header of a **for** statement (section 12.6.3).

NOTE These are the only restricted productions in the grammar:

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

ContinueStatement :
 continue [no *LineTerminator* here] *Identifier*_{opt} ;

BreakStatement :
 break [no *LineTerminator* here] *Identifier*_{opt} ;

ReturnStatement :
 return [no *LineTerminator* here] *Expression*_{opt} ;

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

The practical effect of these restricted productions is as follows:

- When a `++` or `--` token is encountered where the parser would treat it as a postfix operator, and at least one *LineTerminator* occurred between the preceding token and the `++` or `--` token, then a semicolon is automatically inserted before the `++` or `--` token.
- When a `continue`, `break`, `return`, or `throw` token is encountered and a *LineTerminator* is encountered before the next token, a semicolon is automatically inserted after the `continue`, `break`, `return`, or `throw` token.

The resulting practical advice to ECMAScript programmers is:

- A postfix `++` or `--` operator should appear on the same line as its operand.
- An *Expression* in a `return` or `throw` statement should start on the same line as the `return` or `throw` token.
- A label in a `break` or `continue` statement should be on the same line as the `break` or `continue` token.

7.9.2 Examples of Automatic Semicolon Insertion

The source

```
{ 1 2 } 3
```

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

```
{ 1
 2 } 3
```

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

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

which is a valid ECMAScript sentence.

The source

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

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

The source

```
return
a + b
```

is transformed by automatic semicolon insertion into the following:

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

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

The source

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

is transformed by automatic semicolon insertion into the following:

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

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

The source

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

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

The source

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

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

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

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

8 Types

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 as 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 has the value **undefined**.

8.2 The Null Type

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

8.3 The Boolean Type

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

8.4 The String Type

The String type is the set of all finite ordered sequences of zero or more 16-bit unsigned integer values ("elements"). The String type is generally used to represent textual data in a running ECMAScript program, in which case each element in the string is treated as a code point value (see section 6). Each element is regarded as occupying a position within the sequence. These positions are indexed with nonnegative integers. The first element (if any) is at position 0, the next element (if any) at position 1, and so on. The length of a string is the number of elements (i.e., 16-bit values) within it. The empty string has length zero and therefore contains no elements.

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

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

8.5 The Number Type

The Number type has exactly 18437736874454810627 (that is, $2^{64}-2^{53}+3$) values, representing the double-precision 64-bit format IEEE 754 values as specified in the IEEE Standard for Binary Floating-Point Arithmetic, except that the 9007199254740990 (that is, $2^{53}-2$) distinct “Not-a-Number” values of the IEEE Standard are represented in ECMAScript as a single special **NaN** value. (Note that the **NaN** value is produced by the program expression **NaN**, 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

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

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

The remaining 9007199254740990 (that is, $2^{53}-2$) values are denormalised, having the form

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

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

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

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

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

Some ECMAScript operators deal only with integers in the range -2^{31} through $2^{31}-1$, inclusive, or in the range 0 through $2^{32}-1$, inclusive. These operators accept any value of the Number type but first convert each such value to one of 2^{32} integer values. See the descriptions of the ToInt32 and ToUint32 operators in 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 host environment; therefore “ReadOnly” does not mean “constant and unchanging”!)
DontEnum	The property is not to be enumerated by a for-in enumeration (section 12.6.4).
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 part of the language. They are defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon internal properties in the manner described here. For the purposes of this document, the names of internal properties are enclosed in double square brackets `[[ ]]`. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a **TypeError** exception is thrown.

There are two types of access for normal (non-internal) 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 visible 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]]	(<i>PropertyName</i>)	Returns the value of the property.
[[Put]]	(<i>PropertyName</i> , <i>Value</i>)	Sets the specified property to <i>Value</i> .
[[CanPut]]	(<i>PropertyName</i>)	Returns a boolean value indicating whether a [[Put]] operation with <i>PropertyName</i> will succeed.
[[HasProperty]]	(<i>PropertyName</i>)	Returns a boolean value indicating whether the object already has a member with the given name.
[[Delete]]	(<i>PropertyName</i>)	Removes the specified property from the object.
[[DefaultValue]]	(<i>Hint</i>)	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 <i>constructors</i> .
[[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 <i>functions</i> .
[[HasInstance]]	(<i>Value</i>)	Returns a boolean value indicating whether <i>Value</i> delegates behaviour to this object. Of the native ECMAScript objects, only Function objects implement [[HasInstance]].
[[Scope]]	none	A scope chain that defines the environment in which a Function object is executed.
[[Match]]	(<i>String</i> , <i>Index</i>)	Tests for a regular expression match and returns a MatchResult value (see section 15.10.2.1).

Every object (including host objects) must implement the [[Prototype]] and [[Class]] properties and the [[Get]], [[Put]], [[CanPut]], [[HasProperty]], [[Delete]], and [[DefaultValue]] methods. (Note, however, that the [[DefaultValue]] method may, for some objects, simply throw a **TypeError** exception.)

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. The value of a [[Class]] property is used internally to distinguish different kinds of built-in objects. Note that this specification does not provide any means for a program to access that value except through **Object.prototype.toString** (see section 15.2.4.2).

For native objects the [[Get]], [[Put]], [[CanPut]], [[HasProperty]], [[Delete]] and [[DefaultValue]] methods behave as described in described in sections 8.6.2.1, 8.6.2.2, 8.6.2.3, 8.6.2.4, 8.6.2.5 and 8.6.2.6, 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 unless specified otherwise; for example, one possibility is that [[Get]] and [[Put]] for a particular host object indeed fetch and store property values but [[HasProperty]] always generates **false**.

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. Call the **[[CanPut]]** method of **[[Prototype]]** of *O* with property name *P*.
6. Return Result(5).

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. Throw a **TypeError** exception.

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. Throw a **TypeError** exception.

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 (section 15.9), in which case it behaves as if the hint were String.

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

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 access the components of references:

- GetBase(V). Returns the base object component of the reference V.
- GetPropertyName(V). Returns the property name component of the reference V.

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

8.7.1 GetValue (V)

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

8.7.2 PutValue (V, W)

1. If Type(V) is not Reference, throw a **ReferenceError** exception.
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**, **return** and **throw**) that perform nonlocal transfers of control. Values of the Completion type are triples of the form (*type*, *value*, *target*), where *type* is one of **normal**, **break**, **continue**, **return**, or **throw**, *value* is any ECMAScript value or **empty**, and *target* is any ECMAScript identifier or **empty**.

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

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 argument *PreferredType*. 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:

<i>Input Type</i>	<i>Result</i>
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 <code>[[DefaultValue]]</code> method of the object, passing the optional hint <i>PreferredType</i> . The behaviour of the <code>[[DefaultValue]]</code> method is defined by this specification for all native ECMAScript objects (section 8.6.2.6).

9.2 ToBoolean

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

<i>Input Type</i>	<i>Result</i>
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:

<i>Input Type</i>	<i>Result</i>
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>
<NBSP>
<FF>
<VT>
<CR>
<LF>
<LS>
<PS>
<USP>

StrNumericLiteral :::

StrDecimalLiteral
HexIntegerLiteral

StrDecimalLiteral :::

StrUnsignedDecimalLiteral
+ *StrUnsignedDecimalLiteral*
- *StrUnsignedDecimalLiteral*

StrUnsignedDecimalLiteral :::

Infinity
DecimalDigits . *DecimalDigits*_{opt} *ExponentPart*_{opt}
. *DecimalDigits* *ExponentPart*_{opt}
DecimalDigits *ExponentPart*_{opt}

DecimalDigits :::

DecimalDigit
DecimalDigits *DecimalDigit*

DecimalDigit ::: **one of**

0 1 2 3 4 5 6 7 8 9

ExponentPart :::

ExponentIndicator *SignedInteger*

ExponentIndicator ::: **one of**

e E

SignedInteger :::

DecimalDigits
+ *DecimalDigits*
- *DecimalDigits*

HexIntegerLiteral :::

0x *HexDigit*
0x *HexDigit*
HexIntegerLiteral *HexDigit*

HexDigit ::: **one of**

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

Some differences should be noted between the syntax of a *StringNumericLiteral* and a *NumericLiteral* (section 7.8.3):

- A *StringNumericLiteral* may be preceded and/or followed by white space and/or line terminators.
- A *StringNumericLiteral* that is decimal may have any number of leading 0 digits.
- A *StringNumericLiteral* that is decimal may be preceded by + or - to indicate its sign.
- A *StringNumericLiteral* that is empty or contains only white space is converted to +0.

The conversion of a string to a number value is similar overall to the determination of the number value for a numeric literal (section 7.8.3), but some of the details are different, so the process for converting a string numeric literal to a value of Number type is given here in full. This value is determined in two steps: first, a mathematical value (MV) is derived from the string numeric literal; second, this mathematical value is rounded as described below.

- The MV of *StringNumericLiteral* ::: [empty] is 0.
- The MV of *StringNumericLiteral* ::: *StrWhiteSpace* is 0.
- The MV of *StringNumericLiteral* ::: *StrWhiteSpace*_{opt} *StrNumericLiteral* *StrWhiteSpace*_{opt} is the MV of *StrNumericLiteral*, no matter whether white space is present or not.
- The MV of *StrNumericLiteral* ::: *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* ::: *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StrDecimalLiteral* ::: *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral* ::: + *StrUnsignedDecimalLiteral* is the MV of *StrUnsignedDecimalLiteral*.
- The MV of *StrDecimalLiteral* ::: - *StrUnsignedDecimalLiteral* is the negative of the MV of *StrUnsignedDecimalLiteral*. (Note that if the MV of *StrUnsignedDecimalLiteral* is 0, the negative of this MV is also 0. The rounding rule described below handles the conversion of this sign less mathematical zero to a floating-point +0 or -0 as appropriate.)
- The MV of *StrUnsignedDecimalLiteral* ::: **Infinity** is 10^{10000} (a value so large that it will round to $+\infty$).
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* . is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* . *DecimalDigits* is the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n}), where n is the number of characters in the second *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* . *ExponentPart* is the MV of *DecimalDigits* times 10^e , where e is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* . *DecimalDigits* *ExponentPart* is (the MV of the first *DecimalDigits* plus (the MV of the second *DecimalDigits* times 10^{-n})) times 10^e , where n is the number of characters in the second *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* ::: . *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n} , where n is the number of characters in *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* ::: . *DecimalDigits* *ExponentPart* is the MV of *DecimalDigits* times 10^{e-n} , where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *StrUnsignedDecimalLiteral* ::: *DecimalDigits* *ExponentPart* is the MV of *DecimalDigits* times 10^e , where e is the MV of *ExponentPart*.
- The MV of *DecimalDigits* ::: *DecimalDigit* is the MV of *DecimalDigit*.
- The MV of *DecimalDigits* ::: *DecimalDigits* *DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart* ::: *ExponentIndicator* *SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger* ::: *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* ::: + *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* ::: - *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* ::: 0 or of *HexDigit* ::: 0 is 0.
- The MV of *DecimalDigit* ::: 1 or of *HexDigit* ::: 1 is 1.
- The MV of *DecimalDigit* ::: 2 or of *HexDigit* ::: 2 is 2.
- The MV of *DecimalDigit* ::: 3 or of *HexDigit* ::: 3 is 3.
- The MV of *DecimalDigit* ::: 4 or of *HexDigit* ::: 4 is 4.
- The MV of *DecimalDigit* ::: 5 or of *HexDigit* ::: 5 is 5.
- The MV of *DecimalDigit* ::: 6 or of *HexDigit* ::: 6 is 6.
- The MV of *DecimalDigit* ::: 7 or of *HexDigit* ::: 7 is 7.
- The MV of *DecimalDigit* ::: 8 or of *HexDigit* ::: 8 is 8.
- The MV of *DecimalDigit* ::: 9 or of *HexDigit* ::: 9 is 9.
- The MV of *HexDigit* ::: a or of *HexDigit* ::: A is 10.

- The MV of *HexDigit* ::: **b** or of *HexDigit* ::: **B** is 11.
- The MV of *HexDigit* ::: **c** or of *HexDigit* ::: **C** is 12.
- The MV of *HexDigit* ::: **d** or of *HexDigit* ::: **D** is 13.
- The MV of *HexDigit* ::: **e** or of *HexDigit* ::: **E** is 14.
- The MV of *HexDigit* ::: **f** or of *HexDigit* ::: **F** is 15.
- The MV of *HexIntegerLiteral* ::: **0x** *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* ::: **0X** *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* ::: *HexIntegerLiteral* *HexDigit* is (the MV of *HexIntegerLiteral* times 16) plus the MV of *HexDigit*.

Once the exact MV for a string numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0 unless the first non white space character in the string numeric literal is '-', in which case the rounded value is -0. Otherwise, the rounded value must be the number value for the MV (in the sense defined in section 8.5), unless the literal includes a *StrUnsignedDecimalLiteral* and the literal has more than 20 significant digits, in which case the number value may be either the number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th digit position. A digit is *significant* if it is not part of an *ExponentPart* and

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

9.4 ToInteger

The 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 $\text{sign}(\text{Result}(1)) * \text{floor}(\text{abs}(\text{Result}(1)))$.
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 $\text{sign}(\text{Result}(1)) * \text{floor}(\text{abs}(\text{Result}(1)))$.
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 $\text{Result}(4) - 2^{32}$, 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.

$\text{ToInt32}(\text{ToUint32}(x))$ is equal to $\text{ToInt32}(x)$ for all values of x. (It is to preserve this latter property that $+\infty$ and $-\infty$ are mapped to +0.)

ToInt32 maps -0 to +0.

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, $+\infty$, or $-\infty$, return +0.
3. Compute $\text{sign}(\text{Result}(1)) * \text{floor}(\text{abs}(\text{Result}(1)))$.
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 $+\infty$ and $-\infty$ 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, $+\infty$, or $-\infty$, return +0.
3. Compute $\text{sign}(\text{Result}(1)) * \text{floor}(\text{abs}(\text{Result}(1)))$.
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 ToUint16.

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: Call ToPrimitive(input argument, hint String). Call ToString(Result(1)). 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 $k \geq 1$, $10^{k-1} \leq s < 10^k$, the number value for $s \times 10^{n-k}$ is m , and k is as small as possible. Note that k is the number of digits in the decimal representation of s , that s is not divisible by 10, and that the least significant digit of s is not necessarily uniquely determined by these criteria.
6. If $k \leq n \leq 21$, return the string consisting of the k digits of the decimal representation of s (in order, with no leading zeroes), followed by $n-k$ occurrences of the character '0'.
7. If $0 < n \leq 21$, return the string consisting of the most significant n digits of the decimal representation of s , followed by a decimal point '.', followed by the remaining $k-n$ digits of the decimal representation of s .
8. If $-6 < n \leq 0$, return the string consisting of the character '0', followed by a decimal point '.', followed by $-n$ occurrences of the character '0', followed by the k digits of the decimal representation of s .
9. Otherwise, if $k = 1$, return the string consisting of the single digit of s , followed by lowercase character 'e', followed by a plus sign '+' or minus sign '-' according to whether $n-1$ is positive or negative, followed by the decimal representation of the integer $\text{abs}(n-1)$ (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 $\text{abs}(n-1)$ (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 that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 5 be used as a guideline:

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

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:

<i>Input Type</i>	<i>Result</i>
Undefined	Throw a TypeError exception.
Null	Throw a TypeError exception.
Boolean	Create a new Boolean object whose <code>[[value]]</code> property is set to the value of the boolean. See section 15.6 for a description of Boolean objects.
Number	Create a new Number object whose <code>[[value]]</code> property is set to the value of the number. See section 15.7 for a description of Number objects.
String	Create a new String object whose <code>[[value]]</code> property is set to 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 two types of Function objects:

- Program functions are defined in source text by a *FunctionDeclaration* or created dynamically either by using a *FunctionExpression* or by using the built-in **Function** object as a constructor.
- 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 they are excluded from this discussion of execution contexts.

10.1.2 Types of Executable Code

There are three types of ECMAScript executable code:

- *Global code* is source text that is treated as an ECMAScript *Program*. The global code of a particular *Program* does not include any source text that is parsed as part of a *FunctionBody*.
- *Eval code* is the source text supplied to the built-in **eval** function. More precisely, if the parameter to the built-in **eval** function is a string, it is treated as an ECMAScript *Program*. The eval code for a particular invocation of **eval** is the global code portion of the string parameter.
- *Function code* is source text that is parsed as part of a *FunctionBody*. The *function code* of a particular *FunctionBody* does not include any source text that is parsed as part of a nested *FunctionBody*. *Function code* also denotes the source text supplied when using the built-in **Function** object as a constructor. More precisely, the last parameter provided to the **Function** constructor is converted to a string and treated as the *FunctionBody*. If more than one parameter is provided to the **Function** constructor, all parameters except the last one are converted to strings and concatenated together, separated by commas. The resulting string is interpreted as the *FormalParameterList* for the *FunctionBody* defined by the last parameter. The *function code* for a particular instantiation of a **Function** does not include any source text that is parsed as part of a nested *FunctionBody*.

10.1.3 Variable Instantiation

Every execution context has associated with it a variable object. Variables and functions declared in the source text are added as properties of the variable object. For function 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. On entering an execution context, the properties are bound to the variable object in the following order:

- For function code: 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 as arguments to `[[Call]]`. 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 *FunctionDeclaration* in the code, in source text order, create a property of the variable object whose name is the *Identifier* in the *FunctionDeclaration*, whose value is the result returned by creating a Function object as described in section 13, 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. Semantically, this step must follow the creation of *FormalParameterList* properties.
- For each *VariableDeclaration* or *VariableDeclarationNoIn* in the code, create a property of the variable object whose name is the *Identifier* in the *VariableDeclaration* or *VariableDeclarationNoIn*, 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 *FormalParameterList* and *FunctionDeclaration* 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. A scope chain is a list of objects that are searched when evaluating an *Identifier*. When control enters an execution context, a scope chain is created and populated with an initial set of objects, depending on the type of code. During execution within an execution context, the scope chain of the execution context is affected only by **with** statements (section 12.10) and **catch** clauses (section 12.14).

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(1)`, passing the *Identifier* as the property.
3. If `Result(2)` is **true**, return a value of type Reference whose base object is `Result(1)` 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 evaluating 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* (section 15.1), 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, in the HTML document object model the `window` property of the global object is the global object itself.