BS ISO/IEC 16262:2011

BSI Standards Publication

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

bsi.

...making excellence a habit.™

Document Sharing Hub

National foreword

This British Standard is the UK implementation of ISO/IEC 16262:2011. It supersedes BS ISO/IEC 16262:2002 which is withdrawn.

The UK participation in its preparation was entrusted to Technical Committee IST/5, Programming languages, their environments and system software interfaces.

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

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© BSI 2011

ISBN 978 0 580 73487 8

ICS 35.060

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 31 August 2011.

Amendments issued since publication

Date	Text affected

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 16262:2011

BS ISO/IEC 16262:2011

INTERNATIONAL STANDARD

ISO/IEC 16262

Third edition
2011-06.15

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

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

Reference number
ISO/IEC 16262:2011(E)

Documents
Document Sharing Hub

© ISO/IEC 2011

Full document details: BS ISO/IEC 16262:2011

COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2011

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

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

Published in Switzerland

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 16262:2011

Contents

Page

Foreword viii
Introduction ix
1 Scope 1
2 Conformance 1
3 Normative references 1
4 Overview 1
4.1 Web Scripting 2
4.2 Language Overview 2
4.2.1 Objects 3
4.2.2 The Strict Variant of ECMAScript 4
4.3 Terms and definitions 4
5 Notational Conventions 8
5.1 Syntactic and Lexical Grammars 8
5.1.1 Context-Free Grammars 8
5.1.2 The Lexical and RegExp Grammars 8
5.1.3 The Numeric String Grammar 8
5.1.4 The Syntactic Grammar 8
5.1.5 The JSON Grammar 9
5.1.6 Grammar Notation 9
5.2 Algorithm Conventions 12
6 Source Text 13
7 Lexical Conventions 14
7.1 Unicode Format-Control Characters 14
7.2 White Space 15
7.3 Line Terminators 15
7.4 Comments 16
7.5 Tokens 17
7.6 Identifier Names and Identifiers 17
7.6.1 Reserved Words 18
7.7 Punctuators 19
7.8 Literals 20
7.8.1 Null Literals 20
7.8.2 Boolean Literals 20
7.8.3 Numeric Literals 20
7.8.4 String Literals 22
7.8.5 Regular Expression Literals 25
7.9 Automatic Semicolon Insertion 26
7.9.1 Rules of Automatic Semicolon Insertion 26
7.9.2 Examples of Automatic Semicolon Insertion 27
8 Types 28
8.1 The Undefined Type 28
8.2 The Null Type 28
8.3 The Boolean Type 29
8.4 The String Type 29
8.5 The Number Type 29
8.6 The Object Type 30
8.6.1 Property Attributes 30
8.6.2 Object Internal Properties and Methods 31

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

8.7 The Reference Specification Type 35
8.7.1 GetValue (V) 35
8.7.2 PutValue (V, W) 36
8.8 The List Specification Type 36
8.9 The Completion Specification Type 36
8.10 The Property Descriptor and Property Identifier Specification Types 37
8.10.1 IsAccessorDescriptor (Desc) 37
8.10.2 IsDataDescriptor (Desc) 37
8.10.3 IsGenericDescriptor (Desc) 37
8.10.4 FromPropertyDescriptor (Desc) 38
8.10.5 ToPropertyDescriptor (Obj) 38
8.11 The Lexical Environment and Environment Record Specification Types 39
8.12 Algorithms for Object Internal Methods 39
8.12.1 [[GetOwnProperty]] (P) 39
8.12.2 [[GetProperty]] (P) 39
8.12.3 [[Get]] (P) 39
8.12.4 [[CanPut]] (P) 39
8.12.5 [[Put]] (P, V, Throw) 40
8.12.6 [[HasProperty]] (P) 40
8.12.7 [[Delete]] (P, Throw) 41
8.12.8 [[DefaultValue]] (hint) 41
8.12.9 [[DefineOwnProperty]] (P, Desc, Throw) 41

9 Type Conversion and Testing 43
9.1 ToPrimitive 43
9.2 ToBoolean 43
9.3 ToNumber 43
9.3.1 ToNumber Applied to the String Type 44
9.4 ToInteger 46
9.5 ToInt32: (Signed 32 Bit Integer) 47
9.6 ToUint32: (Unsigned 32 Bit Integer) 47
9.7 ToUint16: (Unsigned 16 Bit Integer) 47
9.8 ToString 48
9.8.1 ToString Applied to the Number Type 48
9.9 ToObject 49
9.10 CheckObjectCoercible 49
9.11 IsCallable 49
9.12 The SameValue Algorithm 50

10 Executable Code and Execution Contexts 50
10.1 Types of Executable Code 50
10.1.1 Strict Mode Code 51
10.2 Lexical Environments 51
10.2.1 Environment Records 51
10.2.2 Lexical Environment Operations 56
10.2.3 The Global Environment 56
10.3 Execution Contexts 56
10.3.1 Identifier Resolution 57
10.4 Establishing an Execution Context 57
10.4.1 Entering Global Code 58
10.4.2 Entering Eval Code 58
10.4.3 Entering Function Code 58
10.5 Declaration Binding Instantiation 59
10.6 Arguments Object 60

11 Expressions 63
11.1 Primary Expressions 63
11.1.1 The `this` Keyword 63
11.1.2 Identifier Reference 63
11.1.3 Literal Reference 63
11.1.4 Array Initialiser 63

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

11.1.5 Object Initialiser 65
11.1.6 The Grouping Operator 67
11.2 Left-Hand-Side Expressions 67
11.2.1 Property Accessors 67
11.2.2 The `new` Operator 68
11.2.3 Function Calls 68
11.2.4 Argument Lists 69
11.2.5 Function Expressions 69
11.3 Postfix Expressions 69
11.3.1 Postfix Increment Operator 70
11.3.2 Postfix Decrement Operator 70
11.4 Unary Operators 70
11.4.1 The `delete` Operator 70
11.4.2 The `void` Operator 71
11.4.3 The `typeof` Operator 71
11.4.4 Prefix Increment Operator 71
11.4.5 Prefix Decrement Operator 72
11.4.6 Unary + Operator 72
11.4.7 Unary – Operator 72
11.4.8 Bitwise NOT Operator (~) 72
11.4.9 Logical NOT Operator (!) 73
11.5 Multiplicative Operators 73
11.5.1 Applying the * Operator 73
11.5.2 Applying the / Operator 74
11.5.3 Applying the % Operator 74
11.6 Additive Operators 75
11.6.1 The Addition operator (+) 75
11.6.2 The Subtraction Operator (–) 75
11.6.3 Applying the Additive Operators to Numbers 75
11.7 Bitwise Shift Operators 76
11.7.1 The Left Shift Operator (<<) 76
11.7.2 The Signed Right Shift Operator (>>) 76
11.7.3 The Unsigned Right Shift Operator (>>>) 77
11.8 Relational Operators 77
11.8.1 The Less-than Operator (<) 77
11.8.2 The Greater-than Operator (>) 78
11.8.3 The Less-than-or-equal Operator (<=) 78
11.8.4 The Greater-than-or-equal Operator (>=) 78
11.8.5 The Abstract Relational Comparison Algorithm 78
11.8.6 The instanceof operator 79
11.8.7 The in operator 79
11.9 Equality Operators 80
11.9.1 The Equals Operator (==) 80
11.9.2 The Does-not-equals Operator (!=) 80
11.9.3 The Abstract Equality Comparison Algorithm 80
11.9.4 The Strict Equals Operator (===) 81
11.9.5 The Strict Does-not-equal Operator (!==) 81
11.9.6 The Strict Equality Comparison Algorithm 82
11.10 Binary Bitwise Operators 82
11.11 Binary Logical Operators 83
11.12 Conditional Operator (? :) 84
11.13 Assignment Operators 84
11.13.1 Simple Assignment (=) 85
11.13.2 Compound Assignment (op=) 85
11.14 Comma Operator (,) 85

12 Statements 86
12.1 Block 86

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

12.2 Variable Statement 87
12.2.1 Strict Mode Restrictions 88
12.3 Empty Statement 88
12.4 Expression Statement 89
12.5 The `if` Statement 89
12.6 Iteration Statements 89
12.6.1 The `do-while` Statement 90
12.6.2 The `while` Statement 90
12.6.3 The `for` Statement 90
12.6.4 The `for-in` Statement 91
12.7 The `continue` Statement 92
12.8 The `break` Statement 93
12.9 The `return` Statement 93
12.10 The `with` Statement 93
12.10.1 Strict Mode Restrictions 94
12.11 The `switch` Statement 94
12.12 Labelled Statements 96
12.13 The `throw` Statement 96
12.14 The `try` Statement 96
12.14.1 Strict Mode Restrictions 97
12.15 The `debugger` statement 97

13 Function Definition 98
13.1 Strict Mode Restrictions 99
13.2 Creating Function Objects 99
13.2.1 [[Call]] 100
13.2.2 [[Construct]] 100
13.2.3 The [[ThrowTypeError]] Function Object 100

14 Program 101
14.1 Directive Prologues and the Use Strict Directive 101

15 Standard Built-in ECMAScript Objects 102
15.1 The Global Object 103
15.1.1 Value Properties of the Global Object 103
15.1.2 Function Properties of the Global Object 104
15.1.3 URI Handling Function Properties 105
15.1.4 Constructor Properties of the Global Object 110
15.1.5 Other Properties of the Global Object 111
15.2 Object Objects 111
15.2.1 The Object Constructor Called as a Function 111
15.2.2 The Object Constructor 112
15.2.3 Properties of the Object Constructor 112
15.2.4 Properties of the Object Prototype Object 115
15.2.5 Properties of Object Instances 117
15.3 Function Objects 117
15.3.1 The Function Constructor Called as a Function 117
15.3.2 The Function Constructor 117
15.3.3 Properties of the Function Constructor 118
15.3.4 Properties of the Function Prototype Object 118
15.3.5 Properties of Function Instances 121
15.4 Array Objects 122
15.4.1 The Array Constructor Called as a Function 122
15.4.2 The Array Constructor 123
15.4.3 Properties of the Array Constructor 123
15.4.4 Properties of the Array Prototype Object 124
15.4.5 Properties of Array Instances 140
15.5 String Objects 141
15.5.1 The String Constructor Called as a Function 141
15.5.2 The String Constructor 142

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

15.5.3 Properties of the String Constructor 142
15.5.4 Properties of the String Prototype Object 142
15.5.5 Properties of String Instances 152
15.6 Boolean Objects 152
15.6.1 The Boolean Constructor Called as a Function 152
15.6.2 The Boolean Constructor 152
15.6.3 Properties of the Boolean Constructor 153
15.6.4 Properties of the Boolean Prototype Object 153
15.6.5 Properties of Boolean Instances 154
15.7 Number Objects 154
15.7.1 The Number Constructor Called as a Function 154
15.7.2 The Number Constructor 154
15.7.3 Properties of the Number Constructor 154
15.7.4 Properties of the Number Prototype Object 155
15.7.5 Properties of Number Instances 159
15.8 The Math Object 159
15.8.1 Value Properties of the Math Object 159
15.8.2 Function Properties of the Math Object 161
15.9 Date Objects 165
15.9.1 Overview of Date Objects and Definitions of Abstract Operators 165
15.9.2 The Date Constructor Called as a Function 171
15.9.3 The Date Constructor 171
15.9.4 Properties of the Date Constructor 172
15.9.5 Properties of the Date Prototype Object 173
15.9.6 Properties of Date Instances 181
15.10 RegExp (Regular Expression) Objects 181
15.10.1 Patterns 181
15.10.2 Pattern Semantics 183
15.10.3 The RegExp Constructor Called as a Function 195
15.10.4 The RegExp Constructor 195
15.10.5 Properties of the RegExp Constructor 196
15.10.6 Properties of the RegExp Prototype Object 196
15.10.7 Properties of RegExp Instances 198
15.11 Error Objects 198
15.11.1 The Error Constructor Called as a Function 199
15.11.2 The Error Constructor 199
15.11.3 Properties of the Error Constructor 199
15.11.4 Properties of the Error Prototype Object 199
15.11.5 Properties of Error Instances 200
15.11.6 Native Error Types Used in This Standard 200
15.11.7 *NativeError* Object Structure 201
15.12 The JSON Object 203
15.12.1 The JSON Grammar 203
15.12.2 parse (text [, reviver]) 204
15.12.3 stringify (value [, replacer [, space]]) 206

16 Errors 209

Annex A (informative) **Grammar Summary** 211

Annex B (informative) **Compatibility** 230

Annex C (informative) **The Strict Mode of ECMAScript** 234

Annex D (informative) **Corrections and Clarifications in the 3rd Edition with Possible 2nd Edition Compatibility Impact** 236

Annex E (informative) **Additions and Changes in the 3rd Edition that Introduce Incompatibilities with the 2nd Edition** 237

Bibliography 240

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

Foreword

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

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

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

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

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

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

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

Introduction

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

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

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

The second edition of this International Standard introduced powerful regular expressions, better string handling, new control statements, try/catch exception handling, tighter definition of errors, formatting for numeric output and minor changes in anticipation of forthcoming internationalization facilities and future language growth. The second edition of the ECMAScript standard was published as ISO/IEC 16262 in June 2002.

Since publication of the second edition of ISO/IEC 16262:2002, ECMAScript has achieved massive adoption in conjunction with the World Wide Web where it has become the programming language that is supported by essentially all web browsers. Significant work was done to develop a third edition of ECMAScript. Although that work was not completed and not published as a new edition of ECMAScript, it informs continuing evolution of the language. The present third edition of ISO/IEC 16262 (published as ECMA-262 5th edition) codifies de facto interpretations of the language specification that have become common among browser implementations and adds support for new features that have emerged since the publication of the third edition. Such features include accessor properties, reflective creation and inspection of objects, program control of property attributes, additional array manipulation functions, support for the JSON object encoding format, and a strict mode that provides enhanced error checking and program security.

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

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

Documents
Document Sharing Hub

Full document details: BS ISO/IEC 16262:2011

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

1 Scope

This International Standard defines the ECMAScript scripting language.

2 Conformance

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

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

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

A conforming implementation of ECMAScript is permitted to support program and regular expression syntax not described in this International Standard. In particular, a conforming implementation of ECMAScript is permitted to support program syntax that makes use of the “future reserved words” listed in 7.6.1.2 of this International Standard.

3 Normative references

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

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

4 Overview

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

ECMAScript is an object-oriented programming language for performing computations and manipulating computational objects within a host environment. ECMAScript as defined here is not intended to be computationally self-sufficient; indeed, there are no provisions in this specification for input of external data or

© ISO/IEC 2011 – All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

output of computed results. Instead, it is expected that the computational environment of an ECMAScript program will provide not only the objects and other facilities described in this specification but also certain environment-specific *host* objects, whose description and behaviour are beyond the scope of this specification except to indicate that they may provide certain properties that can be accessed and certain functions that can be called from an ECMAScript program.

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

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

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

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

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

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

4.1 Web Scripting

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

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

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

4.2 Language Overview

The following is an informal overview of ECMAScript—not all parts of the language are described. This overview is not part of the standard proper.

ECMAScript is object-based: basic language and host facilities are provided by objects, and an ECMAScript program is a cluster of communicating objects. An ECMAScript ***object*** is a collection of ***properties***, each with zero or more ***attributes*** that determine how each property can be used—for example, when the Writable attribute for a property is set to **false**, any attempt by executed ECMAScript code to change the value of the property fails. Properties are containers that hold other objects, ***primitive values***, or ***functions***. A primitive value is a member of one of the following built-in types: **Undefined**, **Null**, **Boolean**, **Number**, and **String**; an

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

object is a member of the remaining built-in type **Object**; and a function is a callable object. A function that is associated with an object via a property is a ***method***.

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

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

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

4.2.1 Objects

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

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

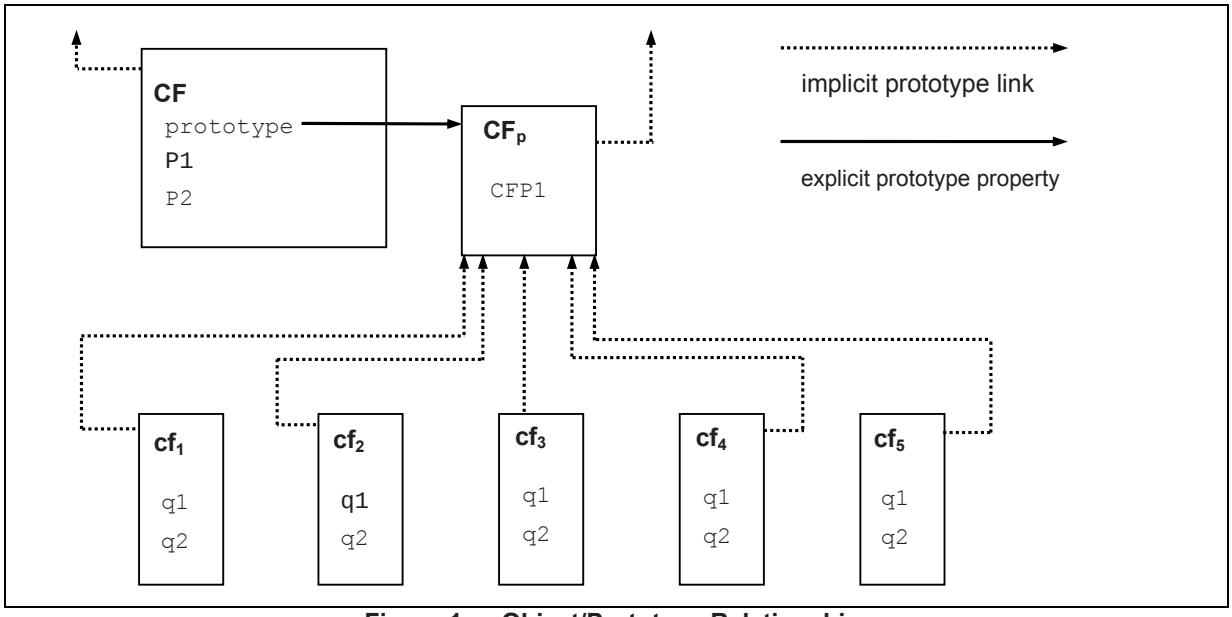

Figure 1 — Object/Prototype Relationships

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

In a class-based object-oriented language, in general, state is carried by instances, methods are carried by classes, and inheritance is only of structure and behaviour. In ECMAScript, the state and methods are carried by objects, and structure, behaviour and state are all inherited.

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

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

Unlike class-based object languages, properties can be added to objects dynamically by assigning values to them. That is, constructors are not required to name or assign values to all or any of the constructed object's properties. In the above diagram, one could add a new shared property for $\mathbf{cf_1}$, $\mathbf{cf_2}$, $\mathbf{cf_3}$, $\mathbf{cf_4}$, and $\mathbf{cf_5}$ by assigning a new value to the property in $\mathbf{CF_p}$.

4.2.2 The Strict Variant of ECMAScript

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

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

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

4.3 Terms and definitions

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

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

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

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

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

4.3.3
object
member of the type Object

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

4.3.4
constructor
function object that creates and initialises objects

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

4.3.5
prototype
object that provides shared properties for other objects

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

4.3.6
native object
object in an ECMAScript implementation whose semantics are fully defined by this specification rather than by the host environment

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

4.3.7
built-in object
object supplied by an ECMAScript implementation, independent of the host environment, that is present at the start of the execution of an ECMAScript program

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

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

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

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

4.3.10
Undefined type
type whose sole value is the undefined value

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

4.3.12
Null type
type whose sole value is the null value

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

4.3.13
Boolean value
member of the Boolean type

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

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

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

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

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

NOTE A String value is a member of the String type. Each integer value in the sequence usually represents a single 16-bit unit of UTF-16 text. However, ECMAScript does not place any restrictions or requirements on the values except that they must be 16-bit unsigned integers.

4.3.17
String type
set of all possible String values

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

NOTE A String object is created by using the `String` constructor in a `new` expression, supplying a String value as an argument. The resulting object has an internal property whose value is the String value. A String object can be coerced to a String value by calling the `String` constructor as a function (15.5.1).

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

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

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

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

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

4.3.22
Infinity
number value that is the positive infinite Number value

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

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

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

NOTE In addition to its named properties, a function contains executable code and state that determine how it behaves when invoked. A function’s code may or may not be written in ECMAScript.

4.3.25
built-in function
built-in object that is a function

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

4.3.26
property
association between a name and a value that is a part of an object

NOTE Depending upon the form of the property, the value may be represented either directly as a data value (a primitive value, an object, or a function object) or indirectly by a pair of accessor functions.

4.3.27
method
function that is the value of a property

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

4.3.28
built-in method
method that is a built-in function

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

4.3.29
attribute
internal value that defines some characteristic of a property

4.3.30
own property
property that is directly contained by its object

4.3.31
inherited property
property of an object that is not an own property but is a property (either own or inherited) of the object’s prototype

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

5 Notational Conventions

5.1 Syntactic and Lexical Grammars

5.1.1 Context-Free Grammars

A *context-free grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of zero or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified alphabet.

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

5.1.2 The Lexical and RegExp Grammars

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

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for ECMAScript and are called ECMAScript *tokens*. These tokens are the reserved words, identifiers, literals, and punctuators of the ECMAScript language. Moreover, line terminators, although not considered to be tokens, also become part of the stream of input elements and guide the process of automatic semicolon insertion (7.9). Simple white space and single-line comments are discarded and do not appear in the stream of input elements for the syntactic grammar. A *MultiLineComment* (that is, a comment of the form "/*…*/" regardless of whether it spans more than one line) is likewise simply discarded if it contains no line terminator; but if a *MultiLineComment* contains one or more line terminators, then it is replaced by a single line terminator, which becomes part of the stream of input elements for the syntactic grammar.

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

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

5.1.3 The Numeric String Grammar

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

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

5.1.4 The Syntactic Grammar

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

When a stream of characters is to be parsed as an ECMAScript program, it is first converted to a stream of input elements by repeated application of the lexical grammar; this stream of input elements is then parsed by a single application of the syntactic grammar. The program is syntactically in error if the tokens in the stream

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

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

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

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

5.1.5 The JSON Grammar

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

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

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

5.1.6 Grammar Notation

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

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

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

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

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

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

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

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

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

is a convenient abbreviation for:

VariableDeclaration **:**
    Identifier
    Identifier Initialiser

and that:

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

is a convenient abbreviation for:

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

which in turn is an abbreviation for:

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

which in turn is an abbreviation for:

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

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

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

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

which is merely a convenient abbreviation for:

© ISO/IEC 2011 — All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

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

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

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

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

DecimalDigits **::**
DecimalDigit
DecimalDigits DecimalDigit

the definition

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

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

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

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

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

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

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

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

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

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

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

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

SourceCharacter ::
any Unicode code unit

5.2 Algorithm Conventions

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

In order to facilitate their use in multiple parts of this specification, some algorithms, called *abstract operations*, are named and written in parameterised functional form so that they may be referenced by name from within other algorithms.

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

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

1. Top-level step
    a. Substep.
    b. Substep
        i. Subsubstep.
        ii. Subsubstep.
            1. Subsubsubstep
                a Subsubsubsubstep

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

A step may specify the iterative application of its substeps.

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

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

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

The mathematical function $\text{sign}(x)$ 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 $\text{floor}(x)$ 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 3.0 or later. The text is expected to have been normalised to Unicode Normalization Form C (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves. ECMAScript source text is assumed to be a sequence of 16-bit code units for the purposes of this specification. Such a source text may include sequences of 16-bit code units that are not valid UTF-16 character encodings. If an actual source text is encoded in a form other than 16-bit code units it must be processed as if it was first converted to UTF-16.

Syntax

SourceCharacter ::
any Unicode code unit

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

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

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

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

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

within a string literal in an ECMAScript program always contributes a character to the String value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

7 Lexical Conventions

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

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

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

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

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

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

Syntax

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

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

7.1 Unicode Format-Control Characters

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

It is useful to allow format-control characters in source text to facilitate editing and display. All format control characters may be used within comments, and within string literals and regular expression literals.

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

<BOM> is a format-control character used primarily at the start of a text to mark it as Unicode and to allow detection of the text's encoding and byte order. <BOM> characters intended for this purpose can sometimes

© ISO/IEC 2011 — All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

also appear after the start of a text, for example as a result of concatenating files. <BOM> characters are treated as white space characters (see 7.2).

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

Table 1 — Format-Control Character Usage

Code Unit Value	*Name*	*Formal Name*	*Usage*
`\u200C`	Zero width non-joiner	<ZWNJ>	*IdentifierPart*
`\u200D`	Zero width joiner	<ZWJ>	*IdentifierPart*
`\uFEFF`	Byte Order Mark	<BOM>	*Whitespace*

7.2 White Space

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

The ECMAScript white space characters are listed in Table 2.

Table 2 — Whitespace Characters

Code Unit Value	*Name*	*Formal Name*
`\u0009`	Tab	<TAB>
`\u000B`	Vertical Tab	<VT>
`\u000C`	Form Feed	<FF>
`\u0020`	Space	<SP>
`\u00A0`	No-break space	<NBSP>
`\uFEFF`	Byte Order Mark	<BOM>
Other category “Zs”	Any other Unicode “space separator”	<USP>

ECMAScript implementations must recognise all of the white space characters defined in Unicode 3.0. Later editions of the Unicode Standard may define other white space characters. ECMAScript implementations may recognise white space characters from later editions of the Unicode Standard.

Syntax

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

7.3 Line Terminators

Like white space characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike white space characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. Line terminators also affect the process of automatic semicolon insertion (7.9). A line terminator

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

cannot occur within any token except a *StringLiteral*. Line terminators may only occur within a *StringLiteral* token as part of a *LineContinuation*.

A line terminator can occur within a *MultiLineComment* (7.4) but cannot occur within a *SingleLineComment*.

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

The ECMAScript line terminator characters are listed in Table 3.

Table 3 — Line Terminator Characters

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

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

Syntax

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

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

7.4 Comments

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

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

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

Syntax

Comment **::**
MultiLineComment
SingleLineComment

© ISO/IEC 2011 — All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

MultiLineComment ::
    /* *MultiLineCommentChars*$_{opt}$ ***/**

MultiLineCommentChars ::
    MultiLineNotAsteriskChar MultiLineCommentChars$_{opt}$
    ***** *PostAsteriskCommentChars*$_{opt}$

PostAsteriskCommentChars ::
    MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars$_{opt}$
    ***** *PostAsteriskCommentChars*$_{opt}$

MultiLineNotAsteriskChar ::
    SourceCharacter **but not ***

MultiLineNotForwardSlashOrAsteriskChar ::
    SourceCharacter **but not one of / or ***

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

SingleLineCommentChars ::
    SingleLineCommentChar SingleLineCommentChars$_{opt}$

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

7.5 Tokens

Syntax

Token ::
    IdentifierName
    Punctuator
    NumericLiteral
    StringLiteral

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

7.6 Identifier Names and Identifiers

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

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

Unicode escape sequences are also permitted in an *IdentifierName*, where they contribute a single character to the *IdentifierName*, as computed by the CV of the *UnicodeEscapeSequence* (see 7.8.4). The **** preceding the *UnicodeEscapeSequence* does not contribute a character to the *IdentifierName*. A *UnicodeEscapeSequence* cannot be used to put a character into an *IdentifierName* that would otherwise be illegal. In other words, if a **** *UnicodeEscapeSequence* sequence were replaced by its *UnicodeEscapeSequence*'s CV, the result must still be a valid *IdentifierName* that has the exact same sequence of characters as the original *IdentifierName*. All interpretations of identifiers within this specification are based upon their actual characters regardless of whether or not an escape sequence was used to contribute any particular characters.

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

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

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

Syntax

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

IdentifierName **::**
    IdentifierStart
    IdentifierName IdentifierPart

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

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

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

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

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

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

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

7.6.1 Reserved Words

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

Syntax

ReservedWord **::**
    Keyword
    FutureReservedWord
    NullLiteral
    BooleanLiteral

© ISO/IEC 2011 — All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

7.6.1.1 Keywords

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

Syntax

Keyword :: **one of**

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

7.6.1.2 Future Reserved Words

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

Syntax

FutureReservedWord **:: one of**

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

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

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

7.7 Punctuators

Syntax

Punctuator **:: one of**

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

DivPunctuator **:: one of**

```
/        /=
```

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

7.8 Literals

Syntax

Literal ::
NullLiteral
BooleanLiteral
NumericLiteral
StringLiteral
RegularExpressionLiteral

7.8.1 Null Literals

Syntax

NullLiteral ::
`null`

Semantics

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

7.8.2 Boolean Literals

Syntax

BooleanLiteral ::
`true`
`false`

Semantics

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

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

7.8.3 Numeric Literals

Syntax

NumericLiteral ::
DecimalLiteral
HexIntegerLiteral

DecimalLiteral ::
DecimalIntegerLiteral `.` $DecimalDigits_{opt}$ $ExponentPart_{opt}$
`.` *DecimalDigits* $ExponentPart_{opt}$
DecimalIntegerLiteral $ExponentPart_{opt}$

DecimalIntegerLiteral ::
`0`
NonZeroDigit $DecimalDigits_{opt}$

DecimalDigits ::
DecimalDigit
DecimalDigits DecimalDigit

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

© ISO/IEC 2011 — All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

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

ExponentPart ::
ExponentIndicator SignedInteger

ExponentIndicator :: **one of**
`e E`

SignedInteger ::
DecimalDigits
+ *DecimalDigits*
– *DecimalDigits*

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

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

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

NOTE For example:

```
3in
```

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

Semantics

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

- The MV of *NumericLiteral* :: *DecimalLiteral* is the MV of *DecimalLiteral*.
- The MV of *NumericLiteral* :: *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* `.` is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* `.` *DecimalDigits* is the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times 10^{-n}), where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* `.` *ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^{e}, where e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* `.` *DecimalDigits ExponentPart* is (the MV of *DecimalIntegerLiteral* plus (the MV of *DecimalDigits* times 10^{-n})) times 10^{e}, where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* :: `.` *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n}, where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalLiteral* :: `.` *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times 10^{e-n}, where n is the number of characters in *DecimalDigits* and e is the MV of *ExponentPart*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral* is the MV of *DecimalIntegerLiteral*.
- The MV of *DecimalLiteral* :: *DecimalIntegerLiteral ExponentPart* is the MV of *DecimalIntegerLiteral* times 10^{e}, where e is the MV of *ExponentPart*.
- The MV of *DecimalIntegerLiteral* :: `0` is 0.
- The MV of *DecimalIntegerLiteral* :: *NonZeroDigit* is the MV of *NonZeroDigit*.
- The MV of *DecimalIntegerLiteral* :: *NonZeroDigit DecimalDigits* is (the MV of *NonZeroDigit* times 10^{n}) plus the MV of *DecimalDigits*, where n is the number of characters in *DecimalDigits*.
- The MV of *DecimalDigits* :: *DecimalDigit* is the MV of *DecimalDigit*.

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

- The MV of *DecimalDigits* :: *DecimalDigits DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart* :: *ExponentIndicator SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger* :: *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* :: + *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* :: – *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* :: `0` or of *HexDigit* :: `0` is 0.
- The MV of *DecimalDigit* :: `1` or of *NonZeroDigit* :: `1` or of *HexDigit* :: `1` is 1.
- The MV of *DecimalDigit* :: `2` or of *NonZeroDigit* :: `2` or of *HexDigit* :: `2` is 2.
- The MV of *DecimalDigit* :: `3` or of *NonZeroDigit* :: `3` or of *HexDigit* :: `3` is 3.
- The MV of *DecimalDigit* :: `4` or of *NonZeroDigit* :: `4` or of *HexDigit* :: `4` is 4.
- The MV of *DecimalDigit* :: `5` or of *NonZeroDigit* :: `5` or of *HexDigit* :: `5` is 5.
- The MV of *DecimalDigit* :: `6` or of *NonZeroDigit* :: `6` or of *HexDigit* :: `6` is 6.
- The MV of *DecimalDigit* :: `7` or of *NonZeroDigit* :: `7` or of *HexDigit* :: `7` is 7.
- The MV of *DecimalDigit* :: `8` or of *NonZeroDigit* :: `8` or of *HexDigit* :: `8` is 8.
- The MV of *DecimalDigit* :: `9` or of *NonZeroDigit* :: `9` or of *HexDigit* :: `9` is 9.
- The MV of *HexDigit* :: `a` or of *HexDigit* :: `A` is 10.
- The MV of *HexDigit* :: `b` or of *HexDigit* :: `B` is 11.
- The MV of *HexDigit* :: `c` or of *HexDigit* :: `C` is 12.
- The MV of *HexDigit* :: `d` or of *HexDigit* :: `D` is 13.
- The MV of *HexDigit* :: `e` or of *HexDigit* :: `E` is 14.
- The MV of *HexDigit* :: `f` or of *HexDigit* :: `F` is 15.
- The MV of *HexIntegerLiteral* :: `0x` *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* :: `0X` *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* :: *HexIntegerLiteral HexDigit* is (the MV of *HexIntegerLiteral* times 16) plus the MV of *HexDigit*.

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

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

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

7.8.4 String Literals

A string literal is zero or more characters enclosed in single or double quotes. Each character may be represented by an escape sequence. All characters may appear literally in a string literal except for the closing quote character, backslash, carriage return, line separator, paragraph separator, and line feed. Any character may appear in the form of an escape sequence.

Syntax

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

© ISO/IEC 2011 — All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

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

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

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

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

LineContinuation **::**
`\` *LineTerminatorSequence*

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

CharacterEscapeSequence **::**
SingleEscapeCharacter
NonEscapeCharacter

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

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

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

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

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

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

Semantics

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

- The SV of *StringLiteral* **::** `""` is the empty character sequence.
- The SV of *StringLiteral* **::** `''` is the empty character sequence.
- The SV of *StringLiteral* **::** `"` *DoubleStringCharacters* `"` is the SV of *DoubleStringCharacters*.

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

- The SV of *StringLiteral* :: ' *SingleStringCharacters* ' is the SV of *SingleStringCharacters*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter* is a sequence of one character, the CV of *DoubleStringCharacter*.
- The SV of *DoubleStringCharacters* :: *DoubleStringCharacter DoubleStringCharacters* is a sequence of the CV of *DoubleStringCharacter* followed by all the characters in the SV of *DoubleStringCharacters* in order.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter* is a sequence of one character, the CV of *SingleStringCharacter*.
- The SV of *SingleStringCharacters* :: *SingleStringCharacter* SingleStringCharacters is a sequence of the CV of *SingleStringCharacter* followed by all the characters in the SV of *SingleStringCharacters* in order.
- The SV of *LineContinuation* :: \ *LineTerminatorSequence* is the empty character sequence.
- The CV of *DoubleStringCharacter* :: *SourceCharacter* **but not one of** " **or** \ **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *DoubleStringCharacter* :: \ *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *DoubleStringCharacter* :: *LineContinuation* is the empty character sequence.
- The CV of *SingleStringCharacter* :: *SourceCharacter* **but not one of** ' **or** \ **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *SingleStringCharacter* :: \ *EscapeSequence* is the CV of the *EscapeSequence*.
- The CV of *SingleStringCharacter* :: *LineContinuation* is the empty character sequence.
- The CV of *EscapeSequence* :: *CharacterEscapeSequence* is the CV of the *CharacterEscapeSequence*.
- The CV of *EscapeSequence* :: **0** [lookahead $\notin$ *DecimalDigit*] is a <NUL> character (Unicode value 0000).
- The CV of *EscapeSequence* :: *HexEscapeSequence* is the CV of the *HexEscapeSequence*.
- The CV of *EscapeSequence* :: *UnicodeEscapeSequence* is the CV of the *UnicodeEscapeSequence*.
- The CV of *CharacterEscapeSequence* :: *SingleEscapeCharacter* is the character whose code unit value is determined by the *SingleEscapeCharacter* according to Table 4:

Table 4 — String Single Character Escape Sequences

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

- The CV of *CharacterEscapeSequence* :: *NonEscapeCharacter* is the CV of the *NonEscapeCharacter*.
- The CV of *NonEscapeCharacter* :: *SourceCharacter* **but not one of** *EscapeCharacter* **or** *LineTerminator* is the *SourceCharacter* character itself.
- The CV of *HexEscapeSequence* :: **x** *HexDigit HexDigit* is the character whose code unit value is (16 times the MV of the first *HexDigit*) plus the MV of the second *HexDigit*.
- The CV of *UnicodeEscapeSequence* :: **u** *HexDigit HexDigit HexDigit HexDigit* is the character whose code unit value is (4096 times the MV of the first *HexDigit*) plus (256 times the MV of the second *HexDigit*) plus (16 times the MV of the third *HexDigit*) plus the MV of the fourth *HexDigit*.

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

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

7.8.5 Regular Expression Literals

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

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

Syntax

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

RegularExpressionBody **::**
  RegularExpressionFirstChar RegularExpressionChars

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

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

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

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

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

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

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

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

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

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

Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

Semantics

A regular expression literal evaluates to a value of the Object type that is an instance of the standard built-in constructor RegExp. This value is determined in two steps: first, the characters comprising the regular expression's *RegularExpressionBody* and *RegularExpressionFlags* production expansions are collected uninterpreted into two Strings Pattern and Flags, respectively. Then each time the literal is evaluated, a new object is created as if by the expression **`new RegExp(`**`Pattern, Flags`**`)`** where RegExp is the standard built-in constructor with that name. The newly constructed object becomes the value of the *RegularExpressionLiteral*. If the call to **`new RegExp`** would generate an error as specified in 15.10.4.1, the error must be treated as an early error (Clause 16).

7.9 Automatic Semicolon Insertion

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

7.9.1 Rules of Automatic Semicolon Insertion

There are three basic rules of semicolon insertion:

1. When, as the program is parsed from left to right, a token (called the *offending token*) is encountered that is not allowed by any production of the grammar, then a semicolon is automatically inserted before the offending token if one or more of the following conditions is true:
   - The offending token is separated from the previous token by at least one *LineTerminator*.
   - The offending token is **`}`**.

2. When, as the program is parsed from left to right, the end of the input stream of tokens is encountered and the parser is unable to parse the input token stream as a single complete ECMAScript *Program*, then a semicolon is automatically inserted at the end of the input stream.

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

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

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

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

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

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

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

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

The practical effect of these restricted productions is as follows:

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

When a **continue**, **break**, **return**, or **throw** token is encountered and a *LineTerminator* is encountered before the next token, a semicolon is automatically inserted after the **continue**, **break**, **return**, or **throw** token.

The resulting practical advice to ECMAScript programmers is:

A postfix **++** or **--** operator should appear on the same line as its operand.

An *Expression* in a **return** or **throw** statement should start on the same line as the **return** or **throw** token.

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

7.9.2 Examples of Automatic Semicolon Insertion

The source

```
{ 1 2 } 3
```

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

```
{ 1
2 } 3
```

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

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

which is a valid ECMAScript sentence.

The source

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

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

The source

```
return
a + b
```

is transformed by automatic semicolon insertion into the following:

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

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

The source

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

is transformed by automatic semicolon insertion into the following:

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

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

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

The source

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

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

The source

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

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

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

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

8 Types

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

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

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

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

8.1 The Undefined Type

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

8.2 The Null Type

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

© ISO/IEC 2011 — All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

8.3 The Boolean Type

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

8.4 The String Type

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

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

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

8.5 The Number Type

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

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

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

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

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

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

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

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

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

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

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

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

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

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

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

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

8.6 The Object Type

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

- A *named data property* associates a name with an ECMAScript language value and a set of Boolean attributes.
- A *named accessor property* associates a name with one or two accessor functions, and a set of Boolean attributes. The accessor functions are used to store or retrieve an ECMAScript language value that is associated with the property.
- An *internal property* has no name and is not directly accessible via ECMAScript language operators. Internal properties exist purely for specification purposes.

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

8.6.1 Property Attributes

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

© ISO/IEC 2011 — All rights reserved
Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

Table 5 — Attributes of a Named Data Property

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

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

Table 6 — Attributes of a Named Accessor Property

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

If the value of an attribute is not explicitly specified by this specification for a named property, the default value defined in Table 7 is used.

Table 7 — Default Attribute Values

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

8.6.2 Object Internal Properties and Methods

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

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

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

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

Table 8 — Internal Properties Common to All Objects

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

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

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

[[Prototype]] depends on the implementation. Every [[Prototype]] chain must have finite length (that is, starting from any object, recursively accessing the [[Prototype]] internal property must eventually lead to a **null** value). Named data properties of the [[Prototype]] object are inherited (are visible as properties of the child object) for the purposes of get access, but not for put access. Named accessor properties are inherited for both get access and put access.

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

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

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

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

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

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

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

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

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

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

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

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

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

Table 9 — Internal Properties Only Defined for Some Objects

Internal Property	*Value Type Domain*	*Description*
[[PrimitiveValue]]	*primitive*	Internal state information associated with this object. Of the standard built-in ECMAScript objects, only Boolean, Date, Number, and String objects implement [[PrimitiveValue]].
[[Construct]]	SpecOp(a List of *any*) → Object	Creates an object. Invoked via the `new` operator. The arguments to the SpecOp are the arguments passed to the **new** operator. Objects that implement this internal method are called *constructors*.
[[Call]]	SpecOp(*any*, a List of *any*) → *any* or Reference	Executes code associated with the object. Invoked via a function call expression. The arguments to the SpecOp are this object and a list containing the arguments passed to the function call expression. Objects that implement this internal method are *callable*. Only callable objects that are host objects may return Reference values.
[[HasInstance]]	SpecOp(*any*) → Boolean	Returns a Boolean value indicating whether the argument is likely an Object that was constructed by this object. Of the standard built-in ECMAScript objects, only Function objects implement [[HasInstance]].
[[Scope]]	Lexical Environment	A lexical environment that defines the environment in which a Function object is executed. Of the standard built-in ECMAScript objects, only Function objects implement [[Scope]].
[[FormalParameters]]	List of Strings	A possibly empty List containing the identifier Strings of a Function's *FormalParameterList*. Of the standard built-in ECMAScript objects, only Function objects implement [[FormalParameterList]].
[[Code]]	ECMAScript code	The ECMAScript code of a function. Of the standard built-in ECMAScript objects, only Function objects implement [[Code]].
[[TargetFunction]]	Object	The target function of a function object created using the standard built-in Function.prototype.bind method. Only ECMAScript objects created using Function.prototype.bind have a [[TargetFunction]] internal property.
[[BoundThis]]	*any*	The pre-bound this value of a function Object created using the standard built-in Function.prototype.bind method. Only ECMAScript objects created using Function.prototype.bind have a [[BoundThis]] internal property.
[[BoundArguments]]	List of *any*	The pre-bound argument values of a function Object created using the standard built-in Function.prototype.bind method. Only ECMAScript objects created using Function.prototype.bind have a [[BoundArguments]] internal property.
[[Match]]	SpecOp(*String*, *index*) → *MatchResult*	Tests for a regular expression match and returns a MatchResult value (see 15.10.2.1). Of the standard built-in ECMAScript objects, only RegExp objects implement [[Match]].
[[ParameterMap]]	Object	Provides a mapping between the properties of an arguments object (see 10.6) and the formal parameters of the associated function. Only ECMAScript objects that are arguments objects have a [[ParameterMap]] internal property.

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

8.7 The Reference Specification Type

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

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

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

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

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

8.7.1 GetValue (V)

1. If Type(*V*) is not Reference, return *V*.
2. Let *base* be the result of calling GetBase(*V*).
3. If IsUnresolvableReference(*V*), throw a **ReferenceError** exception.
4. If IsPropertyReference(*V*), then
    a. If HasPrimitiveBase(*V*) is **false**, then let *get* be the [[Get]] internal method of *base*, otherwise let *get* be the special [[Get]] internal method defined below.
    b. Return the result of calling the *get* internal method using *base* as its **this** value, and passing GetReferencedName(*V*) for the argument.
5. Else, *base* must be an environment record.
    a. Return the result of calling the GetBindingValue (see 10.2.1) concrete method of *base* passing GetReferencedName(*V*) and IsStrictReference(*V*) as arguments.

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

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

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

8.7.2 PutValue (V, W)

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

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

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

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

8.8 The List Specification Type

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

8.9 The Completion Specification Type

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011

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

8.10 The Property Descriptor and Property Identifier Specification Types

The Property Descriptor type is used to explain the manipulation and reification of named property attributes. Values of the Property Descriptor type are records composed of named fields where each field’s name is an attribute name and its value is a corresponding attribute value as specified in 8.6.1. In addition, any field may be present or absent.

Property Descriptor values may be further classified as data property descriptors and accessor property descriptors based upon the existence or use of certain fields. A data property descriptor is one that includes any fields named either [[Value]] or [[Writable]]. An accessor property descriptor is one that includes any fields named either [[Get]] or [[Set]]. Any property descriptor may have fields named [[Enumerable]] and [[Configurable]]. A Property Descriptor value may not be both a data property descriptor and an accessor property descriptor; however, it may be neither. A generic property descriptor is a Property Descriptor value that is neither a data property descriptor nor an accessor property descriptor. A fully populated property descriptor is one that is either an accessor property descriptor or a data property descriptor and that has all of the fields that correspond to the property attributes defined in either 8.6.1 Table 5 or Table 6.

For notational convenience within this specification, an object literal-like syntax can be used to define a property descriptor value. For example, Property Descriptor {[[Value]]: 42, [[Writable]]: **false**, [[Configurable]]: **true**} defines a data property descriptor. Field name order is not significant. Any fields that are not explicitly listed are considered to be absent.

In specification text and algorithms, dot notation may be used to refer to a specific field of a Property Descriptor. For example, if D is a property descriptor then D.[[Value]] is shorthand for “the field of D named [[Value]]”.

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

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

8.10.1 IsAccessorDescriptor (Desc)

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

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

8.10.2 IsDataDescriptor (Desc)

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

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

8.10.3 IsGenericDescriptor (Desc)

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

© ISO/IEC 2011 — All rights reserved

Documents
Document Sharing Hub
Full document details: BS ISO/IEC 16262:2011

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

8.10.4 FromPropertyDescriptor (Desc)

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

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

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

8.10.5 ToPropertyDescriptor (Obj)

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

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

AVA Documents
Document Sharing Hub

© ISO/IEC 2011 — All rights reserved

Full document details: BS ISO/IEC 16262:2011