INTERNATIONAL STANDARD

ISO/IEC 16262

Third edition
2011-06.15

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

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

Reference number
ISO/IEC 16262:2011(E)

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

Published in Switzerland

Contents

Page

Foreword 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

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

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

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

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

Foreword

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

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

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

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

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

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

Introduction

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

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

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

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

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

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

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

1 Scope

This International Standard defines the ECMAScript scripting language.

2 Conformance

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

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

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

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

3 Normative references

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

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

4 Overview

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

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

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

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

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

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

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

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

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

4.1 Web Scripting

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

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

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

4.2 Language Overview

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

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

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

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

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

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

4.2.1 Objects

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

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

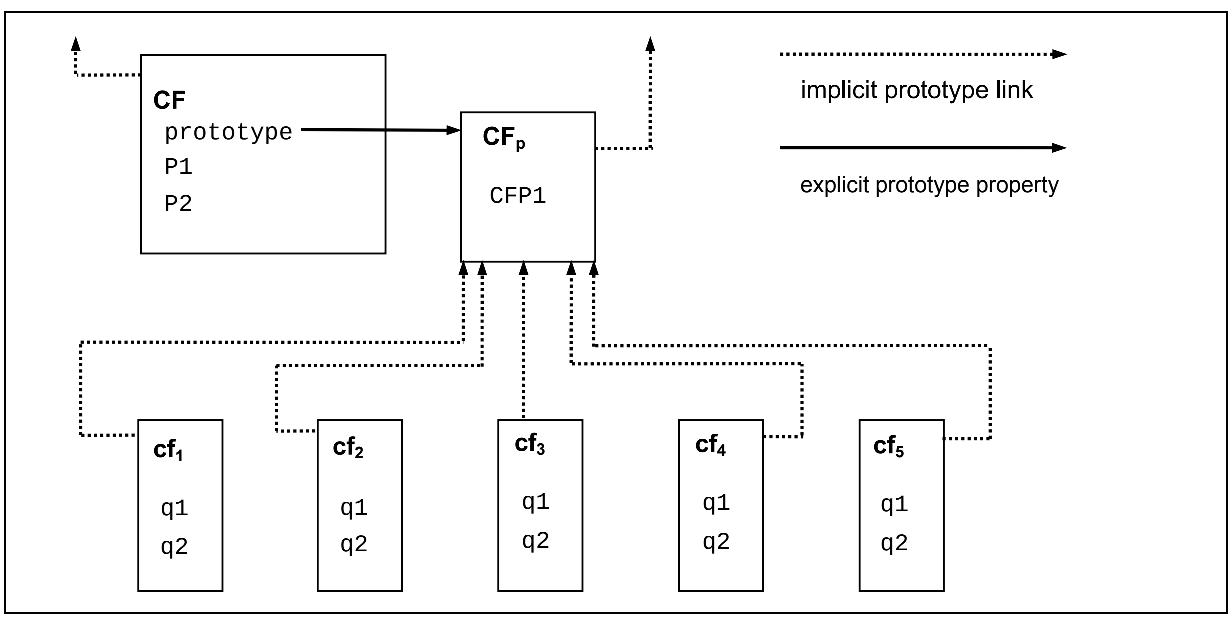

Figure 1 — Object/Prototype Relationships

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

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

CF is a constructor (and also an object). Five objects have been created by using **new** expressions: $\mathbf{cf}_1$, $\mathbf{cf}_2$, $\mathbf{cf}_3$, $\mathbf{cf}_4$, and $\mathbf{cf}_5$. Each of these objects contains properties named `q1` and `q2`. The dashed lines represent the implicit prototype relationship; so, for example, $\mathbf{cf}_3$'s prototype is $\mathbf{CF}_\mathbf{p}$. The constructor, **CF**, has two properties itself, named `P1` and `P2`, which are not visible to $\mathbf{CF}_\mathbf{p}$, $\mathbf{cf}_1$, $\mathbf{cf}_2$, $\mathbf{cf}_3$, $\mathbf{cf}_4$, or $\mathbf{cf}_5$. The property named `CFP1` in $\mathbf{CF}_\mathbf{p}$ 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 $\mathbf{CF}_\mathbf{p}$'s implicit prototype chain that are not named `q1`, `q2`, or `CFP1`. Notice that there is no implicit prototype link between **CF** and $\mathbf{CF}_\mathbf{p}$.

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 $\mathbf{CF}_\mathbf{p}$.

4.2.2 The Strict Variant of ECMAScript

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

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

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

4.3 Terms and definitions

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

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

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

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

4.3.3
object
member of the type Object

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

4.3.4
constructor
function object that creates and initialises objects

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

4.3.5
prototype
object that provides shared properties for other objects

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

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

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

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

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

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

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

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

4.3.10
Undefined type
type whose sole value is the undefined value

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

4.3.12
Null type
type whose sole value is the null value