Brief History

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

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

The third edition of the Standard includes 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 internationalisation facilities and future language growth. The language documented by the third edition has come to be known as ECMAScript 3 or ES3.

Work on the language is not complete. The technical committee is working on significant enhancements, including mechanisms for scripts to be created and used across the Internet, and tighter coordination with other standards bodies such as groups within the World Wide Web Consortium and the Wireless Application Protocol Forum.
### Table of contents

1 Scope  
2 Conformance  
3 References  
4 Overview  
  4.1 Web Scripting  
  4.2 Language Overview  
    4.2.1 Objects  
    4.2.2 The Strict Variant of ECMAScript  
  4.3 Definitions  
    4.3.1 Type  
    4.3.2 Primitive Value  
    4.3.3 Object  
    4.3.4 Constructor  
    4.3.5 Prototype  
    4.3.6 Native Object  
    4.3.7 Built-in Object  
    4.3.8 Host Object  
    4.3.9 Undefined Value  
    4.3.10 Undefined Type  
    4.3.11 Null Value  
    4.3.12 Null Type  
    4.3.13 Boolean Value  
    4.3.14 Boolean Type  
    4.3.15 Boolean Object  
    4.3.16 String Value  
    4.3.17 String Type  
    4.3.18 String Object  
    4.3.19 Number Value  
    4.3.20 Number Type  
    4.3.21 Number Object  
    4.3.22 Infinity  
    4.3.23 NaN  
    4.3.24 Function  
    4.3.25 Built-in Function  
    4.3.26 Property  
    4.3.27 Method  
    4.3.28 Attribute  
    4.3.29 Own Property  
    4.3.30 Inherited Property  
    4.3.31 Built-in Method  
5 Notational Conventions  
  5.1 Syntactic and Lexical Grammars  
    5.1.1 Context-Free Grammars
5.1.2 The Lexical and RegExp Grammars | 7
5.1.3 The Numeric String Grammar | 7
5.1.4 The Syntactic Grammar | 7
5.1.5 Grammar Notation | 8
5.2 Algorithm Conventions | 10

6. Source Text | 11

7 Lexical Conventions | 12
7.1 Unicode Format-Control Characters | 12
7.2 White Space | 12
7.3 Line Terminators | 13
7.4 Comments | 14
7.5 Tokens | 15
7.5.1 Reserved Words | 15
7.5.2 Keywords | 15
7.5.3 Future Reserved Words | 15
7.6 Identifiers | 16
7.7 Punctuators | 17
7.8 Literals | 17
7.8.1 Null Literals | 17
7.8.2 Boolean Literals | 17
7.8.3 Numeric Literals | 17
7.8.4 String Literals | 20
7.8.5 Regular Expression Literals | 22
7.9 Automatic Semicolon Insertion | 23
7.9.1 Rules of Automatic Semicolon Insertion | 23
7.9.2 Examples of Automatic Semicolon Insertion | 24

8 Types | 25
8.1 The Undefined Type | 26
8.2 The Null Type | 26
8.3 The Boolean Type | 26
8.4 The String Type | 26
8.5 The Number Type | 26
8.6 The Object Type | 27
8.6.1 Property Attributes | 27
8.6.2 Object Internal Properties and Methods | 28
8.7 The Reference Specification Type | 31
8.7.1 GetValue (V) | 31
8.7.2 PutValue (V, W) | 31
8.8 The List Specification Type | 31
8.9 The Completion Specification Type | 31
8.10 The Property Descriptor and Property Identifier Specification Types | 32
8.10.1 IsAccessorDescriptor (Desc) | 32
8.10.2 IsDataDescriptor (Desc) | 32
8.10.3 IsGenericDescriptor (Desc) | 32
8.10.4 FromPropertyDescriptor (Desc) | 32
8.10.5  ToPropertyDescriptor ( Desc ) 33
8.11  The Lexical Environment and Environment Record Specification Types 33
8.12  Algorithms for Object Internal Methods 33
  8.12.1  [[GetOwnProperty]] (P) 33
  8.12.2  [[GetProperty]] (P) 34
  8.12.3  [[Get]] (P) 34
  8.12.4  [[CanPut]] (P) 34
  8.12.5  [[ThrowingPut]] ( P, V, Throw ) 34
  8.12.6  [[Put]] (P, V) 35
  8.12.7  [[HasProperty]] (P) 35
  8.12.8  [[Delete]] (P, Throw) 35
  8.12.9  [[DefaultValue]] (hint) 35
  8.12.10  [[DefineOwnProperty]] (P, Desc, Throw) 36

9  Type Conversion and Testing 37
  9.1  ToPrimitive 37
  9.2  ToBoolean 37
  9.3  ToNumber 38
    9.3.1  ToNumber Applied to the String Type 38
  9.4  ToInteger 41
  9.5  ToInt32: (Signed 32 Bit Integer) 41
  9.6  ToUint32: (Unsigned 32 Bit Integer) 41
  9.7  ToUint16: (Unsigned 16 Bit Integer) 42
  9.8  ToString 42
    9.8.1  ToString Applied to the Number Type 42
  9.9  ToObject 43
  9.10  IsCallable 43
  9.11  The SameValue Algorithm 44

10  Executable Code and Execution Contexts 44
  10.1  Types of Executable Code 44
    10.1.1  Strict Mode Code 44
  10.2  Lexical Environments 45
    10.2.1  Environment Records 45
    10.2.2  Lexical Environment Operations 48
    10.2.3  The Global Environment 49
  10.3  Execution Contexts 49
    10.3.1  Identifier Resolution 49
  10.4  Establishing An Execution Context 49
    10.4.1  Global Code 50
    10.4.2  Eval Code 50
    10.4.3  Function Code 50
  10.5  Arguments Object 51
    10.5.1  Strict Mode Restrictions 52
  10.6  Declaration Binding Instantiation 52

11  Expressions 53
  11.1  Primary Expressions 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1.1</td>
<td>The <code>this</code> Keyword</td>
<td>53</td>
</tr>
<tr>
<td>11.1.2</td>
<td>Identifier Reference</td>
<td>53</td>
</tr>
<tr>
<td>11.1.3</td>
<td>Literal Reference</td>
<td>53</td>
</tr>
<tr>
<td>11.1.4</td>
<td>Array Initialiser</td>
<td>53</td>
</tr>
<tr>
<td>11.1.5</td>
<td>Object Initialiser</td>
<td>55</td>
</tr>
<tr>
<td>11.1.6</td>
<td>The Grouping Operator</td>
<td>57</td>
</tr>
<tr>
<td>11.2</td>
<td>Left-Hand-Side Expressions</td>
<td>57</td>
</tr>
<tr>
<td>11.2.1</td>
<td>Property Accessors</td>
<td>57</td>
</tr>
<tr>
<td>11.2.2</td>
<td>The <code>new</code> Operator</td>
<td>58</td>
</tr>
<tr>
<td>11.2.3</td>
<td>Function Calls</td>
<td>58</td>
</tr>
<tr>
<td>11.2.4</td>
<td>Argument Lists</td>
<td>59</td>
</tr>
<tr>
<td>11.2.5</td>
<td>Function Expressions</td>
<td>59</td>
</tr>
<tr>
<td>11.3</td>
<td>Postfix Expressions</td>
<td>59</td>
</tr>
<tr>
<td>11.3.1</td>
<td>Postfix Increment Operator</td>
<td>59</td>
</tr>
<tr>
<td>11.3.2</td>
<td>Postfix Decrement Operator</td>
<td>60</td>
</tr>
<tr>
<td>11.4</td>
<td>Unary Operators</td>
<td>60</td>
</tr>
<tr>
<td>11.4.1</td>
<td>The <code>delete</code> Operator</td>
<td>60</td>
</tr>
<tr>
<td>11.4.2</td>
<td>The <code>void</code> Operator</td>
<td>60</td>
</tr>
<tr>
<td>11.4.3</td>
<td>The <code>typeof</code> Operator</td>
<td>60</td>
</tr>
<tr>
<td>11.4.4</td>
<td>Prefix Increment Operator</td>
<td>61</td>
</tr>
<tr>
<td>11.4.5</td>
<td>Prefix Decrement Operator</td>
<td>61</td>
</tr>
<tr>
<td>11.4.6</td>
<td>Unary <code>+</code> Operator</td>
<td>61</td>
</tr>
<tr>
<td>11.4.7</td>
<td>Unary <code>-</code> Operator</td>
<td>61</td>
</tr>
<tr>
<td>11.4.8</td>
<td>Bitwise NOT Operator (<code>~</code>)</td>
<td>62</td>
</tr>
<tr>
<td>11.4.9</td>
<td>Logical NOT Operator (<code>!</code>)</td>
<td>62</td>
</tr>
<tr>
<td>11.5</td>
<td>Multiplicative Operators</td>
<td>62</td>
</tr>
<tr>
<td>11.5.1</td>
<td>Applying the <code>*</code> Operator</td>
<td>62</td>
</tr>
<tr>
<td>11.5.2</td>
<td>Applying the <code>/</code> Operator</td>
<td>63</td>
</tr>
<tr>
<td>11.5.3</td>
<td>Applying the <code>%</code> Operator</td>
<td>63</td>
</tr>
<tr>
<td>11.6</td>
<td>Additive Operators</td>
<td>64</td>
</tr>
<tr>
<td>11.6.1</td>
<td>The Addition Operator (<code>+</code>)</td>
<td>64</td>
</tr>
<tr>
<td>11.6.2</td>
<td>The Subtraction Operator (<code>-</code>)</td>
<td>64</td>
</tr>
<tr>
<td>11.6.3</td>
<td>Applying the Additive Operators (<code>+</code>, <code>-</code>) to Numbers</td>
<td>64</td>
</tr>
<tr>
<td>11.7</td>
<td>Bitwise Shift Operators</td>
<td>65</td>
</tr>
<tr>
<td>11.7.1</td>
<td>The Left Shift Operator (<code>&lt;&lt;</code>)</td>
<td>65</td>
</tr>
<tr>
<td>11.7.2</td>
<td>The Signed Right Shift Operator (<code>&gt;&gt;</code>)</td>
<td>65</td>
</tr>
<tr>
<td>11.7.3</td>
<td>The Unsigned Right Shift Operator (<code>&gt;&gt;&gt;</code>)</td>
<td>66</td>
</tr>
<tr>
<td>11.8</td>
<td>Relational Operators</td>
<td>66</td>
</tr>
<tr>
<td>11.8.1</td>
<td>The Less-than Operator (<code>&lt;</code>)</td>
<td>66</td>
</tr>
<tr>
<td>11.8.2</td>
<td>The Greater-than Operator (<code>&gt;</code>)</td>
<td>67</td>
</tr>
<tr>
<td>11.8.3</td>
<td>The Less-than-or-equal Operator (<code>&lt;=</code>)</td>
<td>67</td>
</tr>
<tr>
<td>11.8.4</td>
<td>The Greater-than-or-equal Operator (<code>&gt;=</code>)</td>
<td>67</td>
</tr>
<tr>
<td>11.8.5</td>
<td>The Abstract Relational Comparison Algorithm</td>
<td>67</td>
</tr>
<tr>
<td>11.8.6</td>
<td>The <code>instanceof</code> operator</td>
<td>68</td>
</tr>
<tr>
<td>11.8.7</td>
<td>The <code>in</code> operator</td>
<td>68</td>
</tr>
</tbody>
</table>

15 December 2008
11.9 Equality Operators
11.9.1 The Equals Operator (==) 69
11.9.2 The Does-not-equals Operator (!=) 69
11.9.3 The Abstract Equality Comparison Algorithm 69
11.9.4 The Strict Equals Operator (===) 70
11.9.5 The Strict Does-not-equal Operator (!==) 70
11.9.6 The Strict Equality Comparison Algorithm 70
11.10 Binary Bitwise Operators 71
11.11 Binary Logical Operators 71
11.12 Conditional Operator (?:) 72
11.13 Assignment Operators 73
11.13.1 Simple Assignment (=) 73
11.13.2 Compound Assignment (op=) 74
11.14 Comma Operator (,) 74

12 Statements 75
12.1 Block 75
12.2 Variable statement 76
12.3 Empty Statement 77
12.4 Expression Statement 77
12.5 The if Statement 77
12.5.1 Strict Mode Restrictions 78
12.6 Iteration Statements 78
12.6.1 The do-while Statement 78
12.6.2 The while statement 78
12.6.3 The for Statement 79
12.6.4 The for-in Statement 79
12.7 The continue Statement 80
12.8 The break Statement 81
12.9 The return Statement 81
12.10 The with Statement 81
12.10.1 Strict Mode Restrictions 82
12.11 The switch Statement 82
12.11.1 Strict Mode Restrictions 84
12.12 Labelled Statements 84
12.13 The throw statement 84
12.14 The try statement 84
12.15 Debugger statement 85

13 Function Definition 86
13.1 Definitions 87
13.2 Creating Function Objects 87
13.2.1 [[Call]] 87
13.2.2 [[Construct]] 87

14 Program 89
1 Scope
This Standard defines the ECMAScript scripting language.

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

A conforming implementation of this International standard shall interpret characters in conformance with the Unicode Standard, Version 3.0 or later, and ISO/IEC 10646-1 with either UCS-2 or UTF-16 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300. If the adopted encoding form is not otherwise specified, it presumed to be the UTF-16 encoding form.

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

A conforming implementation of ECMAScript is permitted to provide properties not described in this specification, and values for those properties, for objects that are described in this specification.

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

3 References
ISO/IEC 9899:1996 Programming Languages – C, including amendment 1 and technical corrigenda 1 and 2.
ISO/IEC 10646-1:1993 Information Technology -- Universal Multiple-Octet Coded Character Set (UCS) plus its amendments and corrigenda.

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

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

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

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:


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 theWritable 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 methods. A primitive value is a member of one of the following built-in types: Undefined, Null, Boolean, Number, and String; an object is a member of the remaining built-in type Object; and a method is a function associated with an object via a property.

ECMAScript defines a collection of built-in objects that round out the definition of ECMAScript entities. These built-in objects include the Global object, the Object object, the Function object, the Array object, the String object, the 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 contain classes such as those in C++, Smalltalk, or Java, but rather, supports constructors which create objects by executing code that allocates storage for the objects and initialises all or part of them by assigning initial values to their properties. All constructors are objects, but not all objects are constructors. Each constructor has a property named ‘prototype’ that is used to implement prototype-based inheritance and shared properties. Objects are created by using constructors.
ECMAScript supports prototype-based inheritance. Every object created by a constructor has an implicit reference 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.

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

All objects that do not directly contain a particular property that their prototype contains share that property and its value. The following diagram illustrates this:

In 4.2.2 The Strict Variant of ECMAScript

The ECMAScript Language recognizes 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 non-strict ECMAScript language and modifies the detailed semantics of some features. The strict variant

---

Deleted: constructor has an associated prototype, and every
Deleted: that
Deleted: to the prototype
Deleted: associated with its constructor

Deleted: CFP1
Deleted: link
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 Definitions

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

4.3.1 Type

A type is a set of data values as defined in section 8 of this specification.

4.3.2 Primitive Value

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

4.3.3 Object

An object is a member of the type Object. It is a collection of properties.

4.3.4 Constructor

A constructor is a Function object that creates and initialises objects. 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

A prototype is an object used to implement structure, state, and behaviour inheritance in ECMAScript. When a constructor creates an object, that object implicitly references the constructor’s “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.

4.3.6 Native Object

A native object is any object supplied by an ECMAScript implementation independent of the host environment. Standard native objects are defined in this specification. Some native objects are built-in; others may be constructed during the course of execution of an ECMAScript program.

4.3.7 Built-in Object

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

4.3.8 Host Object

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

4.3.9 Undefined Value

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

15 December 2008
4.3.10 Undefined Type
The type **undefined** has exactly one value, called **undefined**.

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

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

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

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

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

4.3.16 String Value
A string value is a primitive value that is a member of the type **String** and is a finite ordered sequence of zero or more 16-bit unsigned integer values.

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

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

4.3.18 String Object
A **String object** is a member of the type **Object** and is an instance of the built-in **String object**. That is, a **String object** is created by using the String constructor in a **new** expression, supplying a string as an argument. The resulting object has an implicit (unnamed) property that is the string. A **String object** can be coerced to a string value by calling the String constructor as a function (15.5.1).

4.3.19 Number Value
A **number value** is a primitive value that is a member of the type **Number** and is a direct representation of a number.

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

4.3.21 Number Object
A **Number object** is a member of the type **Object** and is an instance of the built-in **Number object**. That is, a **Number object** is created by using the Number constructor in a **new** expression, supplying a number as an argument. The resulting object has an implicit (unnamed) property that is the number. A **Number object** can be coerced to a number value by calling the Number constructor as a function (15.7.1).

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

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

4.3.24 **Function**

A function is a member of the type Object that may be invoked as a subroutine. 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**

A built-in function is a function that is a built-in object of the language, such as `parseInt` and `Math.exp`. An implementation may also provide implementation-dependent built-in functions that are not described in this specification.

4.3.26 **Property**

A property is an association between a name and a value. 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) or indirectly by a pair of accessor functions.

4.3.27 **Method**

A method is a function that is the value of a property.

4.3.28 **Attribute**

An attribute is an internal value that defines some characteristic of a property.

4.3.29 **Own Property**

An own property of an object is a property that is directly present on that object.

4.3.30 **Inherited Property**

An inherited property is a property of an object that is not one of its own properties but is a property (either own or inherited) of the object’s prototype.

4.3.31 **Built-in Method**

A built-in method is any method that is a built-in function. Standard built-in methods are defined in this specification, and an ECMAScript implementation may specify and define others. A built-in method is a built-in function.
5 Notational Conventions
5.1 Syntactic and Lexical Grammars

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

5.1.1 Context-Free Grammars

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

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

5.1.2 The Lexical and RegExp Grammars

A lexical grammar for ECMAScript is given in clause 7. This grammar has as its terminal symbols the characters of the Unicode character set. It defines a set of productions, starting from the goal symbol InputElementDiv or InputElementRegExp, that describe how sequences of Unicode characters are translated into a sequence of input elements.

Input elements other than white space and comments form the terminal symbols for the syntactic grammar for ECMAScript and are called ECMAScript tokens. These tokens are the reserved words, identifiers, literals, and punctuators of the ECMAScript language. Moreover, line terminators, although not considered to be tokens, also become part of the stream of input elements and guide the process of automatic semicolon insertion (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 of the Unicode character set. It defines a set of productions, starting from the goal symbol Pattern, that describe how sequences of Unicode characters are translated into regular expression patterns.

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

5.1.3 The Numeric String Grammar

A second grammar is used for translating strings into numeric values. This grammar is similar to the part of the lexical grammar having to do with numeric literals and has as its terminal symbols the characters of the Unicode character set. This grammar appears in 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 Unicode characters is to be parsed as an ECMAScript program, it is first converted to a stream of input elements by repeated application of the lexical grammar; this stream of input elements is then parsed by a single application of the syntax grammar. The program is syntactically in error if the tokens in the stream of input elements cannot be parsed as a single instance of the goal nonterminal Program, with no tokens left over.

15 December 2008
Productions of the syntactic grammar are distinguished by having just one colon “:” as punctuation.

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

5.1.5 Grammar Notation

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

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

\[
\text{WhileStatement} : \text{while} \ (\text{Expression}) \ \text{Statement}
\]

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

\[
\text{ArgumentList} : \ \text{AssignmentExpression} \ \text{ArgumentList} \ \text{AssignmentExpression}
\]

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

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

\[
\text{VariableDeclaration} : \ \text{Identifier Initialiser}_{\text{opt}}
\]

is a convenient abbreviation for:

\[
\text{VariableDeclaration} : \ \text{Identifier} \ \text{Initialiser}
\]

and that:

\[
\text{IterationStatement} : \ \text{for} \ \text{ExpressionNoIn}_{\text{opt}} ; \ \text{Expression}_{\text{opt}} ; \ \text{Expression}_{\text{opt}} \ \text{Statement}
\]

is a convenient abbreviation for:

\[
\text{IterationStatement} : \ \text{for} \ ; \ \text{Expression}_{\text{opt}} ; \ \text{Expression}_{\text{opt}} \ \text{Statement}
\]

which in turn is an abbreviation for:

\[
\text{IterationStatement} : \ \text{for} \ \text{ExpressionNoIn} ; \ \text{Expression}_{\text{opt}} ; \ \text{Expression}_{\text{opt}} \ \text{Statement}
\]

15 December 2008
**IterationStatement**:

\[
\text{for ( ; ; \text{Expression opt}) Statement} \]

\[
\text{for ( ; \text{Expression ; Expression opt}) Statement} \]

\[
\text{for ( ExpressionNoIn ; ; \text{Expression opt}) Statement} \]

\[
\text{for ( ExpressionNoIn ; Expression ; Expression opt) Statement} \]

which in turn is an abbreviation for:

**IterationStatement**:

\[
\text{for ( ; ) Statement} \]

\[
\text{for ( ; \text{Expression}) Statement} \]

\[
\text{for ( ; \text{Expression ;}) Statement} \]

\[
\text{for ( ExpressionNoIn ; ; \text{Expression}) Statement} \]

\[
\text{for ( ExpressionNoIn ; Expression ; ) Statement} \]

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

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

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

**DecimalDigit** :: **one of**

\[
0 1 2 3 4 5 6 7 8 9
\]

**DecimalDigits** ::

\[
\text{DecimalDigit} \\
\text{DecimalDigits DecimalDigit}
\]

the definition

**LookaheadExample** ::

\[
\text{n [lookahead \{1, 3, 5, 7, 9\}] DecimalDigits} \\
\text{DecimalDigit} [\text{lookahead} \{1, 3, 5, 7, 9\}] \]

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

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

**ReturnStatement** :

\[
\text{return [no LineTerminator here] Expressionopt ;}
\]

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

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

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

15 December 2008
NonZeroDigit :: one of
1 2 3 4 5 6 7 8 9

which is merely a convenient abbreviation for:

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

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

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

Identifier ::
IdentifierName but not ReservedWord

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

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

SourceCharacter ::
any Unicode character

5.2 Algorithm Conventions

The specification often uses a numbered list to specify steps in an algorithm. These algorithms are used to
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 parameterized 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”. Type(x) is used as shorthand for “the type of x”.

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 labeled with lower case alphabetic characters and the
second level of substeps labeled 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 a 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 predicate step at the same level.

A step may specify the iterative application of its substeps.

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

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

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

The notation “x modulo y” (y must be finite and nonzero) computes a value k of the same sign as y (or zero) such that abs(k) < abs(y) and x – k = q * y for some integer q.

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

NOTE

floor(x) = x – (x modulo 1).

If an algorithm is defined to “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, using the UTF-16 transformation format. The text is expected to have been normalised to Unicode Normalised Form C (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalisation of text, or behave as though they were performing normalisation of text, themselves.

SourceCharacter ::= any Unicode character

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

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

Within an identifier, the escape sequence contributes one character to the identifier.

NOTE 1

15 December 2008
Although this document sometimes refers to a “transformation” between a “character” within a “string” and the 16-bit unsigned integer that is the UTF-16 encoding of that character, there is actually no transformation because a “character” within a “string” is actually represented using that 16-bit unsigned value.

NOTE 2
ECMAScript differs from the Java programming language in the behaviour of Unicode escape sequences. In a Java program, if the Unicode escape sequence \u000A, for example, occurs within a single-line comment, it is interpreted as a line terminator (Unicode character 000A is line feed) and therefore the next character is not part of the comment. Similarly, if the Unicode escape sequence \u000A occurs within a string literal in a Java program, it is likewise interpreted as a line terminator, which is not allowed within a string literal—one must write \n instead of \u000A to cause a line feed to be part of the string value of a string literal. In an ECMAScript program, a Unicode escape sequence occurring within a comment is never interpreted and therefore cannot contribute to termination of the comment. Similarly, a Unicode escape sequence occurring within a string literal in an ECMAScript program always contributes a character to the string value of the literal and is never interpreted as a line terminator or as a quote mark that might terminate the string literal.

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

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

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

Syntax
InputElementDiv ::=  
    WhiteSpace  
    LineTerminator  
    Comment  
    Token  
    DivPunctuator

InputElementRegExp ::=  
    WhiteSpace  
    LineTerminator  
    Comment  
    Token  
    RegularExpressionLiteral

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

The format control characters may be used in identifiers, within comments, and within string literals and regular expression literals.

7.2 White Space
White space characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other, but are otherwise insignificant. White space may occur between any two tokens, and may occur within strings (where they are considered significant characters forming part of the literal string value), but cannot appear within any other kind of token.
The following characters are considered to be white space:

<table>
<thead>
<tr>
<th>Code Point Value</th>
<th>Name</th>
<th>Formal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\u0009</td>
<td>Tab</td>
<td>&lt;TAB&gt;</td>
</tr>
<tr>
<td>\u000B</td>
<td>Vertical Tab</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>\u000C</td>
<td>Form Feed</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>\u0020</td>
<td>Space</td>
<td>&lt;SP&gt;</td>
</tr>
<tr>
<td>\u0085</td>
<td>Next Line</td>
<td>&lt;NEL&gt;</td>
</tr>
<tr>
<td>\u00A0</td>
<td>No-break space</td>
<td>&lt;NBSP&gt;</td>
</tr>
<tr>
<td>\u200B</td>
<td>Zero width space</td>
<td>&lt;ZWSP&gt;</td>
</tr>
<tr>
<td>\u2028</td>
<td>Line separator</td>
<td>&lt;LS&gt;</td>
</tr>
<tr>
<td>\u2029</td>
<td>Paragraph separator</td>
<td>&lt;PS&gt;</td>
</tr>
</tbody>
</table>

ECMAScript implementations must recognize 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 recognize white space characters from later editions of the Unicode Standard.

Syntax

`WhiteSpace ::= <TAB> | <VT> | <FF> | <SP> | <NEL> | <NBSP> | <ZWSP> | <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. A line terminator cannot occur within any token, except that line terminators that are preceded by an escape sequence may occur within a string literal token. Line terminators also affect the process of automatic semicolon insertion (7.9).

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

The following characters are considered to be line terminators:

<table>
<thead>
<tr>
<th>Code Point Value</th>
<th>Name</th>
<th>Formal Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>\u000A</td>
<td>Line Feed</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>\u000D</td>
<td>Carriage Return</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>\u2028</td>
<td>Line separator</td>
<td>&lt;LS&gt;</td>
</tr>
<tr>
<td>\u2029</td>
<td>Paragraph separator</td>
<td>&lt;PS&gt;</td>
</tr>
</tbody>
</table>
Only the characters in the above table 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

**Description**

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

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

**Syntax**

```
Comment ::=  
MultiLineComment  
SingleLineComment
```

```
MultiLineComment ::=  
/ * MultiLineCommentChars_opt */
```

```
MultiLineCommentChars ::=  
MultiLineNotAsteriskChar MultiLineCommentChars_opt  
* PostAsteriskCommentChars_opt
```

```
PostAsteriskCommentChars ::=  
MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars_opt  
* PostAsteriskCommentChars_opt
```

```
MultiLineNotAsteriskChar ::=  
SourceCharacter but not asterisk *
```

```
MultiLineNotForwardSlashOrAsteriskChar ::=  
SourceCharacter but not forward-slash / or asterisk *
```
SingleLineComment ::
    // SingleLineCommentChars

SingleLineCommentChars ::
    SingleLineCommentChar SingleLineCommentChars opt

SingleLineCommentChar ::
    SourceCharacter but not LineTerminator

7.5 Tokens
Token ::

IdentifierName
Punctuator
NumericLiteral
StringLiteral

7.5.1 Reserved Words
Description
Reserved words cannot be used as identifiers.

Syntax
ReservedWord ::
    Keyword
FutureReservedWord
NullLiteral
BooleanLiteral

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

Syntax
Keyword :: one of

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

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

Syntax
FutureReservedWord :: one of

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

Deleted: ReservedWord Identifier
Deleted: Break
Comment [pL3]: This table needs to be repacked to get rid of the holes.
Deleted: debugger
Note
The identifiers `const`, `let`, and `yield` are likely to be used in a future version of this standard.

### 7.6 Identifiers

**Description**

Identifiers are interpreted according to the grammar given in Section 5.16 of the Unicode standard, with some small modifications. This grammar is based on both normative and informative character categories specified by the Unicode Standard. The characters in the specified categories in version 3.0 of the Unicode 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 identifier.

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

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

ECMAScript implementations may recognize 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 IdentifierPart
IdentifierStart :: UnicodeLetter $ \\
UnicodeEscapeSequence
IdentifierPart :: IdentifierStart UnicodeCombiningMark UnicodeDigit UnicodeConnectorPunctuation UnicodeEscapeSequence

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

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

UnicodeDigit
any character in the Unicode category "Decimal number (Nd)"
```
UnicodeConnectorPunctuation
   any character in the Unicode category “Connector punctuation (Pc)”

UnicodeEscapeSequence
   see 7.8.4.

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

7.7 Punctuators

Syntax
Punctuator :: one of
   { [ ] ( ) ; , < > <= >= != === !== + - * % ++ -- << >> >>> & | ^
   ! ~ && || ? : = += -= *= %= <<= >>= >>>= &= |= ^=

DivPunctuator :: one of
   / /=

7.8 Literals

Syntax
Literal ::
   NullLiteral
   BooleanLiteral
   NumericLiteral
   StringLiteral

7.8.1 Null Literals

Syntax
NullLiteral ::
   null

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

7.8.2 Boolean Literals

Syntax
BooleanLiteral ::
   true
   false

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

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

7.8.3 Numeric Literals

Syntax
NumericLiteral ::
  DecimalLiteral
  HexIntegerLiteral

DecimalLiteral ::
  DecimalIntegerLiteral . DecimalDigits_opt ExponentPart_opt .
  DecimalDigits ExponentPart_opt
  DecimalIntegerLiteral ExponentPart_opt

DecimalIntegerLiteral ::
  0
  NonZeroDigit DecimalDigits_opt

DecimalDigits ::
  DecimalDigit
  DecimalDigits DecimalDigit

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

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

ExponentPart ::
  ExponentIndicator SignedInteger

ExponentIndicator :: one of
  e  E

SignedInteger ::
  DecimalDigits
  + DecimalDigits
  - DecimalDigits

HexIntegerLiteral ::
  0x HexDigit
  0X HexDigit
  HexIntegerLiteral HexDigit

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

NOTE
For example:

3 in

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 ExponentPart 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^-n, where n is the number of characters in DecimalDigits.
The MV of DecimalLiteral :: DecimalIntegerLiteral is the MV of DecimalIntegerLiteral.
The MV of DecimalLiteral :: DecimalIntegerLiteral ExponentPart is the MV of DecimalIntegerLiteral times 10^e, where e is the MV of ExponentPart.
The MV of DecimalIntegerLiteral :: 0 is 0.
The MV of DecimalIntegerLiteral :: NonZeroDigit DecimalDigits is (the MV of NonZeroDigit times 10^n) plus the MV of DecimalDigits, where n is the number of characters in DecimalDigits.
The MV of DecimalDigits :: DecimalDigit is the MV of DecimalDigit.
The MV of DecimalDigits :: DecimalDigits DecimalDigit is (the MV of DecimalDigits times 10) plus the MV of DecimalDigit.
The MV of ExponentPart :: ExponentIndicator SignedInteger is the MV of SignedInteger.
The MV of SignedInteger :: DecimalDigits is the MV of DecimalDigits.
The MV of SignedInteger :: + DecimalDigits is the MV of DecimalDigits.
The MV of SignedInteger :: – DecimalDigits is the negative of the MV of DecimalDigits.
The MV of DecimalDigit :: 0 or of HexDigit :: 0 is 0.
The MV of DecimalDigit :: 1 or of NonZeroDigit :: 1 or of HexDigit :: 1 is 1.
The MV of DecimalDigit :: 2 or of NonZeroDigit :: 2 or of HexDigit :: 2 is 2.
The MV of DecimalDigit :: 3 or of NonZeroDigit :: 3 or of HexDigit :: 3 is 3.
The MV of DecimalDigit :: 4 or of NonZeroDigit :: 4 or of HexDigit :: 4 is 4.
The MV of DecimalDigit :: 5 or of NonZeroDigit :: 5 or of HexDigit :: 5 is 5.
The MV of DecimalDigit :: 6 or of NonZeroDigit :: 6 or of HexDigit :: 6 is 6.
The MV of DecimalDigit :: 7 or of NonZeroDigit :: 7 or of HexDigit :: 7 is 7.
The MV of DecimalDigit :: 8 or of NonZeroDigit :: 8 or of HexDigit :: 8 is 8.
The MV of DecimalDigit :: 9 or of NonZeroDigit :: 9 or of HexDigit :: 9 is 9.
The MV of HexDigit :: a or of HexDigit :: A is 10.
The MV of HexDigit :: b or of HexDigit :: B is 11.
The MV of HexDigit :: c or of HexDigit :: C is 12.
The MV of HexDigit :: d or of HexDigit :: D is 13.
The MV of HexDigit :: e or of HexDigit :: E is 14.
The MV of HexDigit :: 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 (in the sense defined in 8.5), unless the literal is a DecimalLiteral and the literal has more than 20 significant digits, in which case the number value may be either the number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit or the number value for the MV of a literal produced by replacing each significant digit after the 20th with a 0 digit and then incrementing the literal at the 20th significant digit position. A digit is significant if it is not part of an ExponentPart and it is not 0; or there is a nonzero digit to its left and there is a nonzero digit, not in the ExponentPart, to its right.
7.8.4  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 Unicode 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 \\
DoubleStringCharacters ::
  DoubleStringCharacter DoubleStringCharacters opt
SingleStringCharacters ::
  SingleStringCharacter SingleStringCharacters opt
DoubleStringCharacter ::
  SourceCharacter but not double-quote " or backslash \ or LineTerminator \ EscapeSequence \ LineContinuation
SingleStringCharacter ::
  SourceCharacter but not single-quote ' or backslash \ or LineTerminator \ EscapeSequence \ LineContinuation
LineContinuation ::
  \ LineTerminatorSequence
EscapeSequence ::
  CharacterEscapeSequence
  0 (lookahead of DecimalDigit) \ HexEscapeSequence \ UnicodeEscapeSequence
CharacterEscapeSequence ::
  SingleEscapeCharacter NonEscapeCharacter
SingleEscapeCharacter :: one of ' " \ b f n r t v
NonEscapeCharacter ::
  SourceCharacter but not EscapeCharacter or LineTerminator
EscapeCharacter ::
  SingleEscapeCharacter DecimalDigit \ x \ u
HexEscapeSequence ::
  \ HexDigit HexDigit
UnicodeEscapeSequence ::
  \ HexDigit HexDigit HexDigit HexDigit
The definitions of the nonterminal HexDigit is given in section 7.8.3. SourceCharacter is described in sections 2 and 6.

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

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

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Code Point Value</th>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>\u0008</td>
<td>backspace</td>
<td>&lt;BS&gt;</td>
</tr>
<tr>
<td>\t</td>
<td>\u0009</td>
<td>horizontal tab</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>\n</td>
<td>\u000A</td>
<td>line feed (new line)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>\v</td>
<td>\u000B</td>
<td>vertical tab</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>\f</td>
<td>\u000C</td>
<td>form feed</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>\r</td>
<td>\u000D</td>
<td>carriage return</td>
<td>&lt;CR&gt;</td>
</tr>
<tr>
<td>&quot;</td>
<td>\u0022</td>
<td>double quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>'</td>
<td>\u0027</td>
<td>single quote</td>
<td>'</td>
</tr>
</tbody>
</table>
| \\
| \u005C | backslash \ |

The CV of CharacterEscapeSequence :: NonEscapeCharacter is the CV of the NonEscapeCharacter.
The CV of NonEscapeCharacter :: SourceCharacter but not EscapeCharacter or LineTerminator is the SourceCharacter character itself.
The CV of HexEscapeSequence :: x HexDigit HexDigit is the character whose code point value is (16 times the MV of the first HexDigit) plus the MV of the second HexDigit.
The CV of UnicodeEscapeSequence :: u HexDigit HexDigit HexDigit HexDigit is the character whose code
point value is (4096 (that is, 16^2) times the MV of the first HexDigit) plus (256 (that is, 16^1) times the MV of
the second HexDigit) plus (16 times the MV of the third HexDigit) plus the MV of the fourth HexDigit.

NOTE
A line terminator character cannot appear in a string literal, except when preceded by a backslash \ as
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.

7.8.5 Regular Expression Literals
A regular expression literal is an input element that is converted to a RegExp object (section 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 (section 15.10.4) or calling the RegExp
constructor as a function (section 15.10.3).

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

Syntax
RegularExpressionLiteral ::
/ RegularExpressionBody / RegularExpressionFlags
RegularExpressionBody ::
RegularExpressionFirstChar RegularExpressionChars
RegularExpressionChars ::
[empty] RegularExpressionChars RegularExpressionChar
RegularExpressionFirstChar ::
NonTerminator but not * or \ or / or [ 
BackslashSequence
RegularExpressionClass
RegularExpressionChar ::
NonTerminator but not \ or / or [ 
BackslashSequence
RegularExpressionClass
BackslashSequence ::
\ NonTerminator
NonTerminator ::
SourceCharacter but not LineTerminator
RegularExpressionClass ::
[ RegularExpressionClassPreamble RegularExpressionClassChars ]
RegularExpressionClassPreamble ::
[empty]
^ *
RegularExpressionClassChars ::
  [empty]
  RegularExpressionClassChars RegularExpressionClassChar
RegularExpressionClassChar ::
  NonTerminator but not \ or -
  - RegularExpressionClassChar
BackslashExpression
RegularExpressionFlags ::
  [empty]
  RegularExpressionFlags IdentifierPart

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

Semantics
A regular expression literal 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, the error must be reported while scanning the program.

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

7.9.1 Rules of Automatic Semicolon Insertion

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

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

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

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

NOTE
These are the only restricted productions in the grammar:

15 December 2008
PostfixExpression:
  LeftHandSideExpression [noLineTerminator here] ++
  LeftHandSideExpression [noLineTerminator here] --

ContinueStatement:
  continue [noLineTerminator here] Identifier_opt ;

BreakStatement:
  break [noLineTerminator here] Identifier_opt ;

ReturnStatement:
  return [noLineTerminator here] Expression_opt ;

ThrowStatement:
  throw [noLineTerminator here] Expression ;

The practical effect of these restricted productions is as follows:

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

The resulting practical advice to ECMAScript programmers is:

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

7.9.2 Examples of Automatic Semicolon Insertion

The source

{ 1 2 } 3

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

In contrast, the source

{ 1 2 } 3

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

{ 1 ;2 ;} 3;

which is a valid ECMAScript sentence.

The source

for (a; b)

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

The source
return
a + b

is transformed by automatic semicolon insertion into the following:

return;
a + b;

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

The source

\[
a = b
++c
\]

is transformed by automatic semicolon insertion into the following:

\[
a = b;
++c;
\]

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

The source

\[
if \ (a > b)
else c = d
\]

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

The source

\[
a = b + c
(d + e).print()
\]

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

\[
a = b + c(d + e).print()
\]

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

8 Types

Algorithms within this specification manipulate values each of which has an associated type. The possible value types are exactly those defined in this section. 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 artifacts that do not necessarily correspond to any specific entity within an ECMAScript implementation. Specification type values are 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.

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

8.2 The Null Type
The Null type has exactly one value, called null.

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

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

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

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

8.5 The Number Type
The Number type has exactly 18437736874454810627 (that is, 2

\text{18437736874454810627}

) 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

\text{20}

) 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 Non-a-Number values, but such behaviour is implementation-dependent; to ECMAScript code, all NaN values are indistinguishable from each other.

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

The other 18437736874454810624 (that is, 2

\text{18437736874454810624}

) values are called the finite numbers. Half of these are positive numbers and half are negative numbers; for every finite positive number there is a corresponding negative number having the same magnitude.

Note that there is both a positive zero and a negative zero. For brevity, these values are also referred to for expository purposes by the symbols +0 and -0, respectively. (Note that these two zero number values are produced by the program expressions +0 (or simply 0) and -0.)

The 18437736874454810622 (that is, 2

\text{18437736874454810622}

) finite nonzero values are of two kinds:
18428729675200069632 (that is, $2^{54} \cdot 2^{53}$) of them are normalised, having the form

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

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

The remaining 9007199254740990 (that is, $2^{53} \cdot 2$) 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 $\pi$) 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 $+1 \times 2^{31} \times 2^{727}$) and $-2^{1024}$ (which is $-1 \times 2^{31} \times 2^{727}$). 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 $+0$; if $-2^{1024}$ was chosen, replace it with $-0$; if $+0$ was chosen, replace it with $-0$ if and only if $x$ is less than zero; any other chosen value is used unchanged. The result is the number value for $x$. (This procedure corresponds exactly to the behaviour of the IEEE 754 “round to nearest” mode.)

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

### 8.6 The Object Type

An Object is 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 a value and a set of boolean attributes.
- A named accessor property associates a name with a getter method, a setter method, and a set of boolean attributes.
- An internal property has no name and is not directly accessible via the property accessor operators.

Internal properties exist purely for specification purposes. How and when internal properties are used is specified by the language specification below.

There are two types of access for normal (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 following attributes:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Value Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Value]]</td>
<td>Any ECMAScript language type</td>
<td>The value retrieved by reading the property.</td>
</tr>
</tbody>
</table>
8.6.2 Object Internal Properties and Methods

This specification uses various internal properties and methods to define the semantics of object values. These internal properties and methods 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. For the purposes of this document, the names of internal properties are enclosed in double square brackets [[ ]]. When an

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Value]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Getter]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Setter]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Writable]]</td>
<td>false</td>
</tr>
<tr>
<td>[[Enumerable]]</td>
<td>false</td>
</tr>
<tr>
<td>[[Configurable]]</td>
<td>false</td>
</tr>
</tbody>
</table>
algorithm uses an internal property of an object and the object does not implement the indicated internal property, a TypeError exception is thrown.

The following table summarises the internal properties used by this specification that are applicable to all ECMAScript objects. The description indicates their behaviour for native ECMAScript objects, unless stated otherwise in this document for particular types of ECMAScript objects. In particular, Array objects have a slightly different definition of the [[ThrowingPut]] method (see 15.4.5.1) and String objects have a different definition of the [[GetOwnProperty]] method. Host objects may support these internal properties with any implementation-dependent behaviour, or it may be that a host object supports only some internal properties and not others.

The “Value Type Domain” column of the following tables define the types of values associated with internal properties. The type names refer to the types defined in section 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 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 4 Internal Properties Common to All Objects

<table>
<thead>
<tr>
<th>Internal Property</th>
<th>Value Type Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Prototype]]</td>
<td>Object or Null</td>
<td>The prototype of this object.</td>
</tr>
<tr>
<td>[[Class]]</td>
<td>String</td>
<td>A string value indicating a specification defined classification of objects.</td>
</tr>
<tr>
<td>[[PrimitiveValue]]</td>
<td>primitive</td>
<td>Internal state information associated with this object.</td>
</tr>
<tr>
<td>[[Extensible]]</td>
<td>Boolean</td>
<td>If true, own properties may be added to the object.</td>
</tr>
<tr>
<td>[[GetOwnProperty]]</td>
<td>SpecOp(PropertyName) → any</td>
<td>Returns the value of the named property.</td>
</tr>
<tr>
<td>[[GetProperty]]</td>
<td>SpecOp(PropertyName) → Undefined or Property Descriptor</td>
<td>Returns the Property Descriptor of the named own property of this object, or undefined if absent.</td>
</tr>
<tr>
<td>[[CanPut]]</td>
<td>SpecOp(PropertyName) → boolean</td>
<td>Sets the specified named property to the value of the second parameter.</td>
</tr>
<tr>
<td>[[HasProperty]]</td>
<td>SpecOp(PropertyName) → boolean</td>
<td>Returns a boolean value indicating whether a [[Put]] can be performed.</td>
</tr>
<tr>
<td>[[Delete]]</td>
<td>SpecOp(PropertyName, Boolean) → boolean</td>
<td>Removes the specified named own property from the object. The flag controls failure handling.</td>
</tr>
<tr>
<td>[[DefaultValue]]</td>
<td>SpecOp(Hint) → primitive</td>
<td>Hint is a string. Returns a default value for the object.</td>
</tr>
<tr>
<td>[[DefineOwnProperty]]</td>
<td>SpecOp(PropertyName, PropertyDescriptor, Boolean)</td>
<td>Creates or alters the named own property to have the state described by a Property Descriptor. The flag controls failure handling.</td>
</tr>
<tr>
<td>[[ThrowingPut]]</td>
<td>SpecOp(PropertyName, any, Boolean)</td>
<td>Sets the specified named property to the value of the second parameter. The flag controls failure handling.</td>
</tr>
</tbody>
</table>

All ECMAScript objects have an internal property called [[Prototype]]. The value of this property is either null or an object and is used for implementing inheritance. 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.

15 December 2008
Every object (including host objects) must implement the [[Prototype]], [[Class]], and [[Extensible]] internal data properties and the [[Get]], [[GetOwnProperty]], [[GetOwnProperty]], [[DefineOwnProperty]], [[Put]], [[CanPut]], [[HasProperty]], [[Delete]], and [[DefaultValue]] internal methods. (Note, however, that the [[DefaultValue]] method may, for some objects, simply throw a TypeError exception.)

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

The value of the [[Class]] property is defined by this specification for every kind of built-in object. The value of the [[Class]] property of a host object may be any String value, even a value used by a built-in object for its [[Class]] property. The value of a [[Class]] property is used internally to distinguish different kinds of built-in objects. Note that this specification does not provide any means for a program to access that value except through `Object.prototype.toString` (see 15.2.4.2).

For most native objects the common internal methods behave as described in described in § 12, except that Array objects have a slightly different implementation of the [[ThrowingPut]] method (see 15.4.5.1) and String objects have a slightly different implementation of the [[GetOwnProperty]] method (see 15.5.5.2). Host objects may implement these methods in any manner unless specified otherwise; for example, one possibility is that [[Get]] and [[Put]] for a particular host object indeed fetch and store property values but [[HasProperty]] always generates false.

**Table 5 Internal Properties Only Defined for Some Objects**

<table>
<thead>
<tr>
<th>Internal Property</th>
<th>Value Type Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Construct]]</td>
<td>SpecOp(a list of any) → Object</td>
<td>Constructs 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.</td>
</tr>
<tr>
<td>[[Call]]</td>
<td>SpecOp(a list of any) → any or Reference</td>
<td>Executes code associated with the object. Invoked via a function call expression. The arguments to the SpecOp are the arguments passed to the function call expression. Objects that implement this internal method are functions. Only functions that are host objects may return Reference values.</td>
</tr>
<tr>
<td>[[HasInstance]]</td>
<td>SpecOp(any) → Boolean</td>
<td>Returns a Boolean value indicating whether the argument is an Object that delegates behaviour to this object. Of the standard built-in ECMAScript objects, only Objects that are instances of the standard built-in constructor Function implement [[HasInstance]].</td>
</tr>
<tr>
<td>[[Scope]]</td>
<td>Lexical Environment</td>
<td>A lexical environment that defines the environment in which a Function object is executed. Of the standard built-in ECMAScript objects, only Objects that are instances of the standard built-in constructor Function implement [[Scope]].</td>
</tr>
<tr>
<td>[[FormalParameters]]</td>
<td>List</td>
<td>A possibly empty List containing the identifier strings of a Function’s FormalParameterList. Of the standard built-in ECMAScript objects, only Objects that are instances of the standard built-in constructor Function implement [[FormalParameters]].</td>
</tr>
<tr>
<td>[[Code]]</td>
<td>ECMAScript code</td>
<td>The ECMAScript code of a function. Of the standard built-in ECMAScript objects, only Objects that are instances of the standard built-in constructor Function implement [[Code]].</td>
</tr>
<tr>
<td>[[Match]]</td>
<td>SpecOp(string, index) → MatchResult</td>
<td>Tests for a regular expression match and returns a MatchResult value (see section 15.10.2.1). Of the standard built-in ECMAScript objects only objects that are instances of the standard built-in constructor RegExp implement [[Match]].</td>
</tr>
</tbody>
</table>
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 reference to 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 null, an object, or an environment record (10.2.1). A base value of null 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 `true` if the base value is an object and `false` if the base value is an environment record.
- `IsUnresolvableReference(V)`: Returns `true` if the base value is `null` 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. Return the result of calling the `[[Get]]` method of `base`, passing `GetReferencedName(V)` for the argument.
5. Else, `base` must be an environment record.
   a. Return the result of calling the `GetBindingValue(N, S)` concrete method of `Result(2)` passing `GetReferencedName(V)` and `IsStrictReference(V)` as arguments.

### 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 throw a `ReferenceError` exception.
   b. Call the `[[ThrowingPut]]` method for 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. Call the `[[ThrowingPut]]` method of `base`, passing `GetReferencedName(V)` for the property name, W for the value, and `true` for the `Throw` flag.
5. Else `base` must be a reference whose base is an environment record. So,
   a. Call the `SetMutableBinding(N, V, S)` concrete method of `base`, passing `GetReferencedName(V)` for N, W for V, and `IsStrictReference(V)` for S.
6. Return.

### 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.

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. 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 [[Getter]], or [[Setter]]. 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.

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. The order of listing fields names 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 short hand 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.[[Getter]] and Desc.[[Setter]] 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:
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 steps are taken:

The following algorithm assumes that Desc is a fully populated Property Descriptor, such as that returned from [[GetOwnProperty]].
1. If Desc is undefined, then return undefined.
2. Create 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 [[Put]] method of Result(2) with arguments “value” and Desc.[[Value]].
   b. Call the [[Put]] method of Result(2) with arguments “writable” and Desc.[[Writable]].
4. Else, IsAccessorDescriptor(Desc) must be true, so
   a. Call the [[Put]] method of Result(2) with arguments “getter” and Desc.[[Getter]].
   b. Call the [[Put]] method of Result(2) with arguments “setter” and Desc.[[Setter]].
5. Call the [[Put]] method of Result(2) with arguments “enumerable” and Desc.[[Enumerable]].
6. Call the [[Put]] method of Result(2) with arguments “configurable” and Desc.[[Configurable]].
7. Return Result(2).

8.10.5 ToPropertyDescriptor ( Desc )
When the abstract operation ToPropertyDescriptor is called with object Desc, the following steps are taken:
1. If Desc is undefined, then return undefined.
2. Let obj be the result of ToObject(Desc).
3. Let descObj be the result of creating a new Property Descriptor that initially has no fields.
4. If the result of calling the [[HasProperty]] method of obj with argument “enumerable” is true, then
   a. Let enum be the result of calling the [[Get]] method of obj with “enumerable”.
   b. Set the [[Enumerable]] field of descObj to ToBoolean(enum).
5. If the result of calling the [[HasProperty]] method of obj with argument “configurable” is true, then
   a. Let conf be the result of calling the [[Get]] method of obj with argument “configurable”.
   b. Set the [[Configurable]] field of descObj to ToBoolean(conf).
6. If the result of calling the [[HasProperty]] method of obj with argument “value” is true, then
   a. Let value be the result of calling the [[Get]] method of obj with argument “value”.
   b. Set the [[Value]] field of descObj to value.
7. If the result of calling the [[HasProperty]] method of obj with argument “writable” is true, then
   a. Let writable be the result of calling the [[Get]] method of obj with argument “writable”.
   b. Set the [[Writable]] field of descObj to ToBoolean(writable).
8. If the result of calling the [[HasProperty]] method of obj with argument “getter” is true, then
   a. Let getter be the result of calling the [[Get]] method of obj with argument “getter”.
   b. If IsCallable(getter) is false and getter is not undefined, then throw a TypeError exception.
   c. Set the [[Getter]] field of descObj to getter.
9. If the result of calling the [[HasProperty]] method of obj with argument “setter” is true, then
   a. Let setter be the result of calling the [[Get]] method of obj with argument “setter”.
   b. If IsCallable(setter) is false and setter is not undefined, then throw a TypeError exception.
   c. Set the [[Setter]] field of descObj to setter.
10. If either descObj.[[Getter]] or descObj.[[Setter]] are present, then
    a. If either descObj.[[Value]] or descObj.[[Writable]] are present, then throw a TypeError exception.
11. Return descObj.

8.11 The Lexical Environment and Environment Record Specification Types
The Lexical Environment and Environment Record types are used to explain the behaviour of name resolution in nested functions and blocks. These types and the operations upon them are defined in section 10.

8.12 Algorithms for Object Internal Methods
In the following algorithm descriptions, assume O is a native ECMAScript object, P is a string, Desc is a Property Description record, and Throw is a Boolean flag.

8.12.1 [[GetOwnProperty]] (P)
When the [[GetOwnProperty]] internal method of O is called with property name P, the following steps are taken:
1. If O doesn’t have an own property with name P, return undefined.
2. Let D be a newly created Property Descriptor (Section 8.10) with no fields.
3. Let \( X \) be \( O \)'s own property named \( P \).
4. If \( X \) is a data property, then
   a. Set \( D[\text{Value}] \) to the value of \( X \)'s \([\text{Value}]\) attribute.
   b. Set \( D[\text{Writable}] \) to the value of \( X \)'s \([\text{Writable}]\) attribute
5. Else \( X \) is an accessor property, so
   a. Set \( D[\text{Getter}] \) to the value of \( X \)'s \([\text{Getter}]\) attribute.
   b. Set \( D[\text{Setter}] \) to the value of \( X \)'s \([\text{Setter}]\) attribute
6. Set \( D[\text{Enumerable}] \) to the value of \( X \)'s \([\text{Enumerable}]\) attribute.
7. Set \( D[\text{Configurable}] \) to the value of \( X \)'s \([\text{Configurable}]\) attribute.
8. Return \( D \).

Note, however, that if \( O \) is a String object it has a more elaborate \([\text{GetOwnProperty}]\) method (15.5.5.2).

8.12.2 \([\text{GetProperty}]\) (P)
When the \([\text{GetProperty}]\) internal method of \( O \) is called with property name \( P \), the following steps are taken:
1. Let \( prop \) be the result of calling the \([\text{GetOwnProperty}]\) internal method of \( O \) with property name \( P \).
2. If \( prop \) is not \text{undefined}, return \( \text{Result}(1) \).
3. If the \([\text{Prototype}]\) internal property of \( O \) is \text{null}, return \text{undefined}.
4. Call the \([\text{GetProperty}]\) internal method of \([\text{Prototype}]\) with property name \( P \).
5. Return \( \text{Result}(4) \).

8.12.3 \([\text{Get}]\) (P)
When the \([\text{Get}]\) internal method of \( O \) is called with property name \( P \), the following steps are taken:
1. Let \( desc \) be the result of calling the \([\text{GetProperty}]\) internal method of \( O \) with property name \( P \).
2. If \( desc \) is \text{undefined}, return \text{undefined}.
3. If \( \text{IsDataDescriptor}(desc) \) is \text{true}, return \( desc[\text{Value}] \).
4. Otherwise, \( \text{IsAccessorDescriptor}(desc) \) must be \text{true} so, let \( getter \) be \( desc[\text{Getter}] \).
5. If \( getter \) is \text{undefined}, return \text{undefined}.
6. Return the result calling the \([\text{Call}]\) internal method of \( getter \) providing \( O \) as the this value and providing no arguments.

8.12.4 \([\text{CanPut}]\) (P)
When the \([\text{CanPut}]\) internal method of \( O \) is called with property name \( P \), the following steps are taken:
1. Let \( desc \) be the result of calling the \([\text{GetOwnProperty}]\) internal method of \( O \) with argument \( P \).
2. If \( desc \) is not \text{undefined}, then
   a. If \( \text{IsAccessorDescriptor}(desc) \) is \text{true}, then
      i. If \( \text{desc}[\text{Setter}] \) is \text{undefined}, then return \text{false}.
      ii. Else return \text{true}.
   b. Else, \( desc \) must be a Data Descriptor so return the value of \( desc[\text{Writable}] \).
3. Let \( proto \) be the internal \([\text{Prototype}]\) internal property of \( O \).
4. If \( proto \) is \text{null}, then return the value of the \([\text{Extensible}]\) internal property of \( O \).
5. Let \( inherited \) be the result of calling the \([\text{GetProperty}]\) internal method of \( proto \) with property name \( P \).
6. If \( inherited \) is \text{undefined}, return the value of the \([\text{Extensible}]\) internal property of \( O \).
7. If \( \text{IsAccessorDescriptor}(inherited) \) is \text{true}, then
   a. If \( \text{inherited}[\text{Setter}] \) is \text{undefined}, then return \text{false}.
   b. Else return \text{true}.
8. Else, \( inherited \) must be a Data Descriptor
   a. If the \([\text{Extensible}]\) internal property of \( O \) is \text{false}, return \text{false}.
   b. Else return the value of \( inherited[\text{Writable}] \).

\text{NOTE}
Host objects may define additional constraints upon \([\text{Put}]\) operations. If possible, host objects should not allow \([\text{Put}]\) operations in situations where this definition of \([\text{CanPut}]\) returns \text{false}.

8.12.5 \([\text{ThrowingPut}]\) (P, V, Throw)
When the \([\text{ThrowingPut}]\) internal method of \( O \) is called with property \( P \), value \( V \), and boolean flag \( \text{Throw} \) the following steps are taken:

15 December 2008
1. 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.
2. Let `ownDesc` be the result of calling the `[[GetOwnProperty]]` method of `O` with argument `P`.
3. If `IsDataDescriptor(ownDesc)` is `true`, then
   a. Set the `[[Value]]` attribute of property `P` of `O` to `V`.
   b. Return.
4. Let `desc` be the result of calling the `[[GetProperty]]` method of `O` with argument `P`. This may be either an own or inherited accessor property descriptor or an inherited data property descriptor.
5. If `IsAccessorDescriptor(desc)` is `true`, then
   a. Let `setter` be `desc`.`[[Setter]]` which cannot be `undefined`.
   b. Call the `[[Call]]` method of `setter` providing `O` as the this value and providing `V` as the sole argument.
6. Else, create a named data property named `P` on object `O` whose attributes are:
   a. `[[Value]]`: `V`,
   b. `[[Writable]]`: `true`,
   c. `[[Enumerable]]`: `true`,
   d. `[[Configurable]]`: `true`.
7. Return.

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

8.2.6 `[[Put]]` (P, V)
`[[Put]]` is primarily used in the specification of built-in methods. Algorithms that require explicit control over the handling of invalid property stores should call `[[ThrowingPut]]` directly.

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

2. Return.

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

   Let `desc` be the result of calling the `[[GetProperty]]` internal method of `O` with property name `P`.
   If `desc` is `undefined`, then return `false`.
   Else return `true`.

8.2.8 `[[Delete]]` (P, Throw)
When the `[[Delete]]` internal method of `O` is called with property name `P` and the boolean flag `Throw`, the following steps are taken:

1. Let `desc` be the result of calling the `[[GetOwnProperty]]` internal method of `O` with property name `P`.
2. If `desc` is `undefined`, then return `true`.
3. If `desc` `[[Configurable]]` is `true`, then
   a. Remove the own property with name `P` from `O`.
   b. Return `true`.
4. Else if `Throw`, then throw a `TypeError` exception.
5. Return `false`.

8.2.9 `[[DefaultValue]]` (hint)
When the `[[DefaultValue]]` internal method of `O` is called with hint `String`, the following steps are taken:

1. Let `toString` be the result of calling the `[[Get]]` internal method of object `O` with argument "toString".
2. If `toString` is an object then,
3. Let `str` be the result of calling the `[[Call]]` internal method of `toString`, with `O` as the this value and an empty argument list.
4. If `str` is a primitive value, return `str`.
5. Let `valueOf` be the result of calling the `[[Get]]` internal method of object `O` with argument "valueOf".
6. If `valueOf` is an object then,
7. Let val be the result of calling the [[Call]] internal method of `valueOf`, with `O` as the this value and an empty argument list.
8. If `val` is a primitive value, return `val`.
9. Throw a `TypeError` exception.

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

1. Let `valueOf` be the result of calling the `[[Get]]` internal method of object `O` with argument “`valueOf`”.
2. If `valueOf` is an object then,
   a. Let `val` be the result of calling the `[[Call]]` internal method of `valueOf`, with `O` as the this value and an empty argument list.
   b. If `val` is a primitive value, return `val`.
3. Let `toString` be the result of calling the `[[Get]]` internal method of object `O` with argument “`toString`”.
4. If `toString` is an object then,
   a. Let `str` be the result of calling the `[[Get]]` internal method of `toString`, with `O` as the this value and an empty argument list.
   b. If `str` is a primitive value, return `str`.
5. Throw a `TypeError` exception.

When the `[[DefaultValue]]` internal method of `O` is called with no hint, it behaves as if the hint were `Number`, unless `O` is a Date object (see 15.9), in which case it behaves as if the hint were `String`.

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

### 8.12.10 `[[DefineOwnProperty]]` (P, Desc, Throw)

In the following algorithm, the term “Reject” means “If `Throw` is `true`, then throw a `TypeError` exception, otherwise return.”

When the `[[DefineOwnProperty]]` internal method of `O` is called with property name `P`, property descriptor `Desc`, and boolean flag `Throw`, the following steps are taken:

1. Let `current` be the result of calling the `[[GetOwnProperty]]` internal method of `O` with property name `P`.
2. Let `extensible` be the value of the `[[Extensible]]` internal property of `O`.
3. If `current` is `undefined` and `extensible` is `false`, then Reject.
4. If `current` is `undefined` and `extensible` is `true`, then
   a. If `IsGenericDescriptor(Desc)` or `IsDataDescriptor(Desc)` is `true`, then
      i. Create an own data property named `P` of object `O` whose [[Value]], [[Writable]], [[Enumerable]] and [[Configurable]] attribute values are described by `Desc`. If the value of an attribute field of `Desc` is absent, the attribute of the newly created property is set to its default value.
   b. Else, `Desc` must be an accessor Property Descriptor so,
      i. Create an own accessor property named `P` of object `O` whose [[Getter]], [[Setter]], [[Enumerable]] and [[Configurable]] attribute values are described by `Desc`. If the value of an attribute field of `Desc` is absent, the attribute of the newly created property is set to its default value.
   c. Return.
5. Return, if every field in `Desc` is absent.
6. Return, if every field in `Desc` also occurs in `current` and the value of every field in `Desc` is the same value as the corresponding field in `current`.
7. If the `[[Configurable]]` field of `current` is `false` then
   a. Reject, if the `[[Configurable]]` field of `Desc` is `true`.
   b. Reject, if the `[[Enumerable]]` field of `current` and `Desc` are the Boolean negation of each other.
8. If `IsGenericDescriptor(Desc)` is `true`, then no further validation is required.
9. Else, if `IsDataDescriptor(current)` and `IsDataDescriptor(Desc)` have different results, then
   a. Reject, if the `[[Configurable]]` field of `current` is `false`.
   b. If `IsDataDescriptor(current)` is `true`, then
      i. Convert the property named `P` of object `O` from a data property to an accessor property. Preserve the existing values of the converted property’s
[Configurable] and [Enumerable] attributes and set the rest of the property’s attributes to their default values.

c. Else,
   i. Convert the property named \( P \) of object \( O \) from an accessor property to a data property. Preserve the existing values of the converted property’s [Configurable] and [Enumerable] attributes and set the rest of the property’s attributes to their default values.

10. Else, if IsDataDescriptor(\( current \)) and IsDataDescriptor(\( Desc \)) are both true, then
   a. If the [Configurable] field of \( current \) is \( false \), then
      i. Reject, if the [Writable] field of \( current \) is \( false \) and the [Writable] field of \( Desc \) is \( true \).
      ii. If the [Writable] field of \( current \) is \( true \), then
          1. Reject, if the [Value] field of \( Desc \) is present and SameValue(\( Desc \).[Value], \( current \).[Value]) is \( false \).
   b. Else, the [Configurable] field of \( current \) is \( false \) so any change is acceptable.

11. Else, IsAccessorDescriptor(\( current \)) and IsAccessorDescriptor(\( Desc \)) are both true, so,
   a. If the [Configurable] field of \( current \) is \( false \), then
      i. Reject, if the [Setter] field of \( Desc \) is present and SameValue(\( Desc \).[Setter], \( current \).[Setter]) is \( false \).
      ii. Reject, if the [Getter] field of \( Desc \) is present and SameValue(\( Desc \).[Getter], \( current \).[Getter]) is \( false \).

12. For each attribute field of \( Desc \) that is present, set the correspondingly named attribute of the property named \( P \) of object \( O \) to the value of the field.

13. Return.

9 Type Conversion and Testing

The ECMAScript runtime system performs automatic type conversion as needed. To clarify the semantics of certain constructs it is useful to define a set of conversion abstract operators. These abstract operators are not a part of the language; they are defined here to aid the specification of the semantics of the language. The conversion abstract operators are polymorphic; that is, they can accept a value of any ECMAScript language type, but not of specification types.

9.1 ToPrimitive

The abstract operator ToPrimitive takes a Value argument and an optional argument PreferredType. The conversion abstract operators are polymorphic; that is, they can accept a value of any ECMAScript language type, but not of specification types.

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>The result equals the input argument (no conversion).</td>
</tr>
<tr>
<td>Null</td>
<td>The result equals the input argument (no conversion).</td>
</tr>
<tr>
<td>Boolean</td>
<td>The result equals the input argument (no conversion).</td>
</tr>
<tr>
<td>Number</td>
<td>The result equals the input argument (no conversion).</td>
</tr>
<tr>
<td>String</td>
<td>The result equals the input argument (no conversion).</td>
</tr>
<tr>
<td>Object</td>
<td>Return a default value for the Object. The default value of an object is retrieved by calling the internal [[DefaultValue]] method of the object, passing the optional hint PreferredType. The behaviour of the [[DefaultValue]] method is defined by this specification for all native ECMAScript objects (8.6.2.6).</td>
</tr>
</tbody>
</table>

9.2 ToBoolean

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

15 December 2008
### 9.3 ToNumber

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

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>false</td>
</tr>
<tr>
<td>Null</td>
<td>false</td>
</tr>
<tr>
<td>Boolean</td>
<td>The result equals the input argument (no conversion).</td>
</tr>
<tr>
<td>Number</td>
<td>The result is false if the argument is +0, -0, or NaN; otherwise the result is true.</td>
</tr>
<tr>
<td>String</td>
<td>The result is false if the argument is the empty string (its length is zero); otherwise the result is true.</td>
</tr>
<tr>
<td>Object</td>
<td>true</td>
</tr>
</tbody>
</table>

#### 9.3.1 ToNumber Applied to the String Type

ToNumber applied to strings applies the following grammar to the input string. If the grammar cannot interpret the string as an expansion of **StringNumericLiteral**, then the result of ToNumber is **NaN**.

```
StringNumericLiteral :::
  StrWhiteSpace opt
    StrWhiteSpace opt
      StrNumericLiteral StrWhiteSpace opt

StrWhiteSpace :::
  StrWhiteSpaceChar StrWhiteSpace opt

StrWhiteSpaceChar :::
  WhiteSpace
  LineTerminator

StrNumericLiteral :::
  StrDecimalLiteral
  HexIntegerLiteral

StrDecimalLiteral :::
  StrUnsignedDecimalLiteral
  + StrUnsignedDecimalLiteral
  - StrUnsignedDecimalLiteral
```

**Deleted:** `<TAB>`
* `<SP>`
* `<NBSP>`
* `<FF>`
* `<VT>`
* `<CR>`
* `<LF>`
* `<LS>`
* `<PS>`
Some differences should be noted between the syntax of a StringNumericLiteral and a NumericLiteral (see 7.8.3):

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

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

- The MV of StringNumericLiteral ::: [empty] is 0.
- The MV of StringNumericLiteral ::: StrWhiteSpace is 0.
- The MV of StringNumericLiteral ::: StrWhiteSpace StrNumericLiteral StrWhiteSpace is the MV of StrNumericLiteral, no matter whether white space is present or not.
- The MV of StrNumericLiteral ::: StrDecimalLiteral is the MV of StrDecimalLiteral.
- The MV of StrNumericLiteral ::: HexIntegerLiteral is the MV of HexIntegerLiteral.
- The MV of StrDecimalLiteral ::: StrUnsignedDecimalLiteral is the MV of StrUnsignedDecimalLiteral.
- The MV of StrDecimalLiteral ::: + StrUnsignedDecimalLiteral is the MV of StrUnsignedDecimalLiteral.
- The MV of StrDecimalLiteral ::: − StrUnsignedDecimalLiteral is the negative of the MV of StrUnsignedDecimalLiteral. (Note that if the MV of StrUnsignedDecimalLiteral is 0, the negative of
this MV is also 0. The rounding rule described below handles the conversion of this sign less mathematical zero to a floating-point +0 or -0 as appropriate.

- The MV of StrUnsignedDecimalLiteral::: Infinity is 10^1000 (a value so large that it will round to +\infty).
- The MV of StrUnsignedDecimalLiteral::: DecimalDigits . is the MV of DecimalDigits.
- The MV of StrUnsignedDecimalLiteral::: DecimalDigits . DecimalDigits is the MV of the first DecimalDigits plus (the MV of the second DecimalDigits times 10^n), where n is the number of characters in the second DecimalDigits.

The MV of StrUnsignedDecimalLiteral::: DecimalDigits . ExponentPart is the MV of DecimalDigits times 10^e, where e is the MV of ExponentPart.

The MV of StrUnsignedDecimalLiteral::: DecimalDigits . DecimalDigits ExponentPart is (the MV of the first DecimalDigits plus (the MV of the second DecimalDigits times 10^n)) times 10^e, where n is the number of characters in the second DecimalDigits and e is the MV of ExponentPart.

The MV of StrUnsignedDecimalLiteral::: DecimalDigits is the MV of DecimalDigits times 10^n, where n is the number of characters in DecimalDigits.

The MV of StrUnsignedDecimalLiteral::: DecimalDigits ExponentPart is the MV of DecimalDigits times 10^e, where n is the number of characters in DecimalDigits and e is the MV of ExponentPart.

The MV of StrUnsignedDecimalLiteral::: DecimalDigits is the MV of DecimalDigits.

The MV of StrUnsignedDecimalLiteral::: DecimalDigits ExponentPart is the MV of DecimalDigits times 10^e, where e is the MV of ExponentPart.

The MV of DecimalDigits::: DecimalDigits is the MV of DecimalDigit.

The MV of DecimalDigits::: DecimalDigits DecimalDigits 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 ExponentPart is the MV of DecimalDigit.

The MV of SignedInteger::: DecimalDigits is the negative of the MV of DecimalDigits.

The MV of DecimalDigit::: 0 or of HexDigit::: 0 is 0.

The MV of DecimalDigit::: 1 or of HexDigit::: 1 is 1.

The MV of DecimalDigit::: 2 or of HexDigit::: 2 is 2.

The MV of DecimalDigit::: 3 or of HexDigit::: 3 is 3.

The MV of DecimalDigit::: 4 or of HexDigit::: 4 is 4.

The MV of DecimalDigit::: 5 or of HexDigit::: 5 is 5.

The MV of DecimalDigit::: 6 or of HexDigit::: 6 is 6.

The MV of DecimalDigit::: 7 or of HexDigit::: 7 is 7.

The MV of DecimalDigit::: 8 or of HexDigit::: 8 is 8.

The MV of DecimalDigit::: 9 or of HexDigit::: 9 is 9.

The MV of HexDigit::: a or of HexDigit::: A is 10.

The MV of HexDigit::: b or of HexDigit::: B is 11.

The MV of HexDigit::: c or of HexDigit::: C is 12.

The MV of HexDigit::: d or of HexDigit::: D is 13.

The MV of HexDigit::: e or of HexDigit::: E is 14.

The MV of HexDigit::: f or of HexDigit::: F is 15.

The MV of HexIntegerLiteral::: 0X HexDigit is the MV of HexDigit.

The MV of HexIntegerLiteral::: 0X HexDigit is the MV of HexDigit.

The MV of HexIntegerLiteral::: 0X HexDigit is (the MV of HexIntegerLiteral times 16) plus the MV of HexDigit.

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

9.4 ToInteger
The abstract operator ToInteger converts its argument to an integral numeric value. This operator functions
as follows:
1. Call ToNumber on the input argument.
2. If Result(1) is NaN, return +0.
3. If Result(1) is +0, -0, +∞, or -∞, return Result(1).
4. Compute sign(Result(1)) * floor(abs(Result(1))).
5. Return Result(4).

9.5 ToInt32: (Signed 32 Bit Integer)
The abstract operator ToInt32 converts its argument to one of 2^{32} integer values in the range -2^{31} through
2^{31}-1, inclusive. This abstract operator functions as follows:
1. Call ToNumber on the input argument.
2. If Result(1) is NaN, +0, -0, +∞, or -∞, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. Compute Result(3) modulo 2^{32}, that is, a finite integer value k of Number type with positive sign and
   less than 2^{32} in magnitude such the mathematical difference of Result(3) and k is mathematically an
   integer multiple of 2^{32}.
5. If Result(4) is greater than or equal to 2^{31}, return Result(4) - 2^{32}, otherwise return Result(4).

NOTE
Given the above definition of ToInt32:
The ToInt32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves
that value unchanged.

ToInt32(ToInt32(x)) is equal to ToInt32(x) for all values of x. (It is to preserve this latter property that +∞ and
−∞ are mapped to +0.)

9.6 ToUint32: (Unsigned 32 Bit Integer)
The abstract operator ToUint32 converts its argument to one of 2^{32} integer values in the range 0 through
2^{32}-1, inclusive. This abstract operator functions as follows:
1. Call ToNumber on the input argument.
2. If Result(1) is NaN, +0, -0, +∞, or -∞, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. Compute Result(3) modulo 2^{32}, that is, a finite integer value k of Number type with positive sign and
   less than 2^{32} in magnitude such the mathematical difference of Result(3) and k is mathematically an
   integer multiple of 2^{32}.
5. Return Result(4).

NOTE
Given the above definition of ToUint32:
Step 5 is the only difference between ToUint32 and ToInt32.
The ToUint32 abstract operation is idempotent: if applied to a result that it produced, the second application leaves
that value unchanged.

ToUint32(ToInt32(x)) is equal to ToUint32(x) for all values of x. (It is to preserve this latter property that +∞ and
−∞ are mapped to +0.)

15 December 2008
ToUint32 maps -0 to +0.

9.7 ToUint16: (Unsigned 16 Bit Integer)
The abstract operator ToUint16 converts its argument to one of 2¹⁶ integer values in the range 0 through 2¹⁶-1, inclusive. This abstract operator functions as follows:
1. Call ToNumber on the input argument.
2. If Result(1) is NaN, +0, -0, +∞, or -∞, return +0.
3. Compute sign(Result(1)) * floor(abs(Result(1))).
4. Compute Result(3) modulo 2¹⁶; that is, a finite integer value k of Number type with positive sign and less than 2¹⁶ in magnitude such the mathematical difference of Result(3) and k is mathematically an integer multiple of 2¹⁶.
5. Return Result(4).

NOTE
Given the above definition of ToUint16:
The substitution of 2¹⁶ for 2³² in step 4 is the only difference between ToUint32 and ToUint16.
ToUint16 maps -0 to +0.

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

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;null&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>If the argument is true, then the result is &quot;true&quot;. If the argument is false, then the result is &quot;false&quot;.</td>
</tr>
<tr>
<td>Number</td>
<td>See note below.</td>
</tr>
<tr>
<td>String</td>
<td>Return the input argument (no conversion)</td>
</tr>
<tr>
<td>Object</td>
<td>Apply the following steps:</td>
</tr>
<tr>
<td></td>
<td>1. Call ToPrimitive(input argument, hint String).</td>
</tr>
<tr>
<td></td>
<td>2. Call ToString(Result(1)).</td>
</tr>
<tr>
<td></td>
<td>3. Return Result(2).</td>
</tr>
</tbody>
</table>

9.8.1 ToString Applied to the Number Type
The abstract operator ToString converts a number m to string format as follows:
1. If m is NaN, return the string "NaN".
2. If m is +0 or -0, return the string "0".
3. If m is less than zero, return the string concatenation of the string "-" and ToString(-m).
4. If m is infinity, return the string "Infinity".
5. Otherwise, let n, k, and s be integers such that 1 ≤ k ≤ 10ⁿ⁺¹ ≤ s ≤ 10⁶, the number value for s x 10⁻ⁿ is m, and k is as small as possible. Note that k is the number of digits in the decimal representation of s, that s is not divisible by 10, and that the least significant digit of s is not necessarily uniquely determined by these criteria.
6. If k ≤ n ≤ 21, return the string consisting of the k digits of the decimal representation of s (in order, with no leading zeroes), followed by n·k occurrences of the character ‘0’.
7. If 0 < n ≤ 21, return the string consisting of the most significant n digits of the decimal representation of s, followed by a decimal point ‘.’, followed by the remaining k-n digits of the decimal representation of s.
8. If \(-6 < n \leq 0\), return the string consisting of the character ‘0’, followed by a decimal point ‘.’, followed by \(-n\) occurrences of the character ‘0’, followed by the \(k\) digits of the decimal representation of \(s\).

9. Otherwise, if \(k = 1\), return the string consisting of the single digit of \(s\), followed by lowercase character ‘e’, followed by a plus sign ‘+’ or minus sign ‘-’ according to whether \(n-1\) is positive or negative, followed by the decimal representation of the integer \(\text{abs}(n-1)\) (with no leading zeros).

10. Return the string consisting of the most significant digit of the decimal representation of \(s\), followed by a decimal point ‘.’, followed by the remaining \(k-1\) digits of the decimal representation of \(s\), followed by the lowercase character ‘e’, followed by a plus sign ‘+’ or minus sign ‘-’ according to whether \(n-1\) is positive or negative, followed by the decimal representation of the integer \(\text{abs}(n-1)\) (with no leading zeros).

**NOTE**

The following observations may be useful as guidelines for implementations, but are not part of the normative requirements of this Standard:

If \(x\) is any number value other than -0, then ToNumber(ToString(x)) is exactly the same number value as \(x\).

The least significant digit of \(s\) is not always uniquely determined by the requirements listed in step 5.

For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 5 be used as a guideline:

Otherwise, let \(n\), \(k\), and \(s\) be integers such that \(k \geq 1\), \(10^{-k} \leq s < 10^k\), the number value for \(s\times10^{-k}\) is \(m\), and \(k\) is as small as possible. If there are multiple possibilities for \(s\), choose the value of \(s\) for which \(s \times 10^{-k}\) is closest in value to \(m\). If there are two such possible values of \(s\), choose the one that is even. Note that \(k\) is the number of digits in the decimal representation of \(s\) and that \(s\) is not divisible by 10.

Implementors of ECMAScript may find useful the paper and code written by David M. Gay for binary-to-decimal conversion of floating-point numbers:


9.9 ToObject

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

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Null</td>
<td>Throw a TypeError exception.</td>
</tr>
<tr>
<td>Boolean</td>
<td>Create a new Boolean object whose [[PrimitiveValue]] property is set to the value of the boolean. See 15.6 for a description of Boolean objects.</td>
</tr>
<tr>
<td>Number</td>
<td>Create a new Number object whose [[PrimitiveValue]] property is set to the value of the number. See 15.7 for a description of Number objects.</td>
</tr>
<tr>
<td>String</td>
<td>Create a new String object whose [[PrimitiveValue]] property is set to the value of the string. See 15.5 for a description of String objects.</td>
</tr>
<tr>
<td>Object</td>
<td>The result is the input argument (no conversion).</td>
</tr>
</tbody>
</table>

9.10 IsCallable

The abstract operator IsCallable determines if its argument, which must be an ECMAScript language value, is a callable function Object according to the following table:

<table>
<thead>
<tr>
<th>Input Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>return false.</td>
</tr>
<tr>
<td>Null</td>
<td>return false.</td>
</tr>
</tbody>
</table>
9.11 The SameValue Algorithm

The internal comparison abstract operation SameValue(x, y), where x and y are ECMAScript language values, produces true or false. Such a comparison is performed as follows:

1. If Type(x) is different from Type(y), return false.
2. If Type(x) is Undefined, return true.
3. If Type(x) is Null, return true.
4. If Type(x) is Number, then
   a. If x is NaN and y is NaN, return true.
   b. If x is +0 and y is -0, return false.
   c. If x is -0 and y is +0, return false.
   d. If x is the same number value as y, return true.
   e. Return false.
5. If Type(x) is String, then return true if x and y are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return false.
6. If Type(x) is Boolean, return true if x and y are both true or both false; otherwise, return false.
7. Return true if x and y refer to the same object. Otherwise, return false.

10 Executable Code and Execution Contexts

10.1 Types of Executable Code

There are three types of ECMAScript executable code:

Global code is source text that is treated as an ECMAScript Program. The global code of a particular Program does not include any source text that is parsed as part of a FunctionBody.

Eval code is the source text supplied to the built-in eval function. More precisely, if the parameter to the built-in eval function is a string, it is treated as an ECMAScript Program. The eval code for a particular invocation of eval is the global code portion of the string parameter.

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

10.1.1 Strict Mode Code

As described in section 4.2.2, an ECMAScript Program syntactic unit may be processed using either unrestricted or strict mode syntax and semantics. When processed using strict mode the three types of ECMAScript code are referred to as strict global code, strict eval code, and strict function code. Code is interpreted in strict mode code in the following situations:

- Global code is strict global code if the Program that defines the global code includes a UseStrictDirective.
- Eval code is strict eval code if the Program that defines the eval code includes a UseStrictDirective or if the call to eval is a direct call (see section 15.1.2.1) to the eval function that is contained in strict mode code.
● Function code that is part of a FunctionDeclaration or FunctionExpression is strict function code if its FunctionDeclaration or FunctionExpression is contained in strict mode code or if its FunctionBody includes a UseStrictDirective.

● Function code that is supplied as the last argument to the built-in Function constructor is strict function code if the last argument is a string that when processed as a FunctionBody includes a UseStrictDirective.

10.2 Lexical Environments

A Lexical Environment is a specification type used to define the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code. A Lexical Environment consists of an Environment Record and a possibly null reference to an outer Lexical Environment. Usually a Lexical Environment is associated with some specific syntactic structure of ECMAScript code such as a FunctionDeclaration, a WithStatement, or a catch clause of a TryStatement and a new Lexical Environment is created each time such code is evaluated.

An Environment Record records the identifier bindings that are created within the scope of its associated Lexical Environment.

The outer environment reference is used to model the dynamic nesting of Lexical Environment values. The outer reference of a (inner) Lexical Environment is a reference to the Lexical Environment that logically surrounds the inner Lexical Environment. An outer Lexical Environment may, of course, have its own outer Lexical Environment. A Lexical Environment may serve as the outer environment for multiple inner Lexical Environments. For example, if a FunctionDeclaration contains two nested FunctionDeclarations then the Lexical Environments of each of the nested functions will have as their outer Lexical Environment the Lexical Environment of the current execution of the surrounding function.

Lexical Environments and Environment Record values are purely specification mechanisms and need not correspond to any particular artifact of an ECMAScript implementation. It is impossible for an ECMAScript program to directly access or manipulate such values.

10.2.1 Environment Records

There are two kinds of Environment Record values used in this specification: declarative environment records and object environment records. Declarative environment records are used to define the effect of ECMAScript language syntactic elements such as FunctionDeclarations, VariableDeclarations, and Catch clauses that directly associate Identifier bindings with ECMAScript language values or variables. Object environment records are used to define the effect of ECMAScript elements such as Program and WithStatement that associate Identifier bindings with the properties of some object.

For specification purposes Environment Record values can be thought of as existing in a simple object-oriented hierarchy where EnvironmentRecord is an abstract class with two concrete subclasses, declarative environment record and object environment record. The abstract class defines the following abstract specification methods that have distinct concrete algorithms for each of its subclasses:

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasBinding(N)</td>
<td>Determine if an environment record has a binding for an identifier.</td>
</tr>
<tr>
<td></td>
<td>Return true if it does and false if it does not. The string value N is</td>
</tr>
<tr>
<td></td>
<td>the text of the identifier.</td>
</tr>
<tr>
<td>CreateMutableBinding(N)</td>
<td>Create a new mutable binding in an environment record. The string</td>
</tr>
<tr>
<td></td>
<td>value N is the text of the bound name.</td>
</tr>
<tr>
<td>GetBindingValue(N,S)</td>
<td>Returns the value of an already existing binding from an environment</td>
</tr>
<tr>
<td></td>
<td>record. The string value N is the text of the bound name. If S is true</td>
</tr>
<tr>
<td></td>
<td>and the binding is an uninitialized immutable binding throw a Reference</td>
</tr>
<tr>
<td></td>
<td>Error exception. S is used to identify strict mode references.</td>
</tr>
<tr>
<td>SetMutableBinding(N,V,S)</td>
<td>Set the value of an already existing mutable binding in an environment</td>
</tr>
<tr>
<td></td>
<td>record. The string value N is the text of the bound name. V is the</td>
</tr>
<tr>
<td></td>
<td>value for the binding and may be a value of any ECMAScript language type</td>
</tr>
<tr>
<td></td>
<td>S is a Boolean flag. If S is true and the binding can not be set throw</td>
</tr>
<tr>
<td></td>
<td>a TypeError exception. S is used to identify strict mode references.</td>
</tr>
</tbody>
</table>
10.2.1.1 Declarative Environment Records

Each declarative environment record is associated with a ECMAScript program scope containing variable, and or function declarations. A declarative environment record binds the set of identifiers defined by the declarations contained within its scope.

In addition to the mutable binds supported by all Environment Records, declarative environment records also provide for immutable bindings. An immutable binding is one where the association between an identifier and a value may not be modified once it has been established. Declarative environment records support the following methods in addition to the Environment Record abstract specification methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateImmutableBinding(N)</td>
<td>Create a new but uninitialized immutable binding in an environment record. The string value N is the text of the bound name.</td>
</tr>
<tr>
<td>InitializeImmutableBinding(N,V)</td>
<td>Set the value of an already existing but uninitialized immutable binding in an environment record. The string value N is the text of the bound name. V is the value for the binding and is a value of any ECMAScript language type.</td>
</tr>
</tbody>
</table>

The behaviour of the concrete specification methods for Declarative Environment Records are defined by the following algorithms.

10.2.1.1.1 HasBinding(N)

The concrete environment record method HasBinding for declarative environment records simply determines if the argument identifier is one of the identifiers bound by the record:

1. Let envRec be the declarative environment record for which the method was invoked.
2. If envRec has a binding for the name that is the value of N, return true.
3. If it does not have such a binding, return false.

10.2.1.1.2 CreateMutableBinding (N)

The concrete Environment Record method CreateMutableBinding for declarative environment records creates a new mutable binding for the name N that is initialized to the value undefined. A binding must not already exist in this Environment Record for N.

1. Let envRec be the declarative environment record for which the method was invoked.
2. Assert: envRec does not already have a binding for N.
3. Create a mutable binding in envRec for N and set its bound value to undefined.

10.2.1.1.3 SetMutableBinding (N,V,S)

The concrete Environment Record method SetMutableValue for declarative environment records attempts to change the bound value of the current binding of the identifier whose name is the value of the argument N to the value of argument V. A binding for N must already exist. If the binding is an immutable binding, a TypeError is always thrown. The S argument is ignored because strict mode does not change the meaning of setting bindings in declarative environment records have.

1. Let envRec be the declarative environment record for which the method was invoked.
2. Assert: envRec must have a binding for N.
3. If the binding for N in envRec is a mutable binding, change its bound value to V.
4. Else this must be an attempt to change the value of an immutable binding so throw a TypeError exception.

10.2.1.1.4 GetBindingValue(N,S)

The concrete Environment Record method GetBindingValue for declarative environment records simply returns the value of its bound identifier whose name is the value of the argument N. The binding must already exist. If S is true and the binding is an uninitialized immutable binding throw a ReferenceError exception.

1. Let envRec be the declarative environment record for which the method was invoked.
2. Assert: $envRec$ has a binding for $N$.
3. If the binding for $N$ in $envRec$ is an uninitialized immutable binding, then
   a. If $S$ is false, return the value undefined, otherwise throw a ReferenceError exception.
4. Else, return the value currently bound to $N$ in $envRec$.

10.2.1.1.5 CreateImmutableBinding (N)
The concrete Environment Record method CreateImmutableBinding for declarative environment records creates a new immutable binding for the name $N$ that is initialized to the value undefined. A binding must not already exist in this environment record for $N$.
1. Let $envRec$ be the declarative environment record for which the method was invoked.
2. Assert: $envRec$ does not already have a binding for $N$.
3. Create a immutable binding in $envRec$ for $N$ and record that it is uninitialized.

10.2.1.1.6 InitializeImmutableBinding (N,V)
The concrete Environment Record method InitializeImmutableBinding for declarative environment records is used to set the bound value of the current binding of the identifier whose name is the value of the argument $N$ to the value of argument $V$. A uninitialized immutable binding for $N$ must already exist.
1. Let $envRec$ be the declarative environment record for which the method was invoked.
2. Assert: $envRec$ must have an unitialized immutable binding for $N$.
3. Set the bound value for $N$ in $envRec$ to $V$.
4. Record that the immutable binding for $N$ in $envRec$ has been initialized.

10.2.1.2 Object Environment Records
Each object environment record is associated with an object called its binding object. An environment record binds the set of identifiers that directly correspond to the property names of its binding object. Property names that are not identifiers are not included in the set of bound identifiers. Because properties can be dynamically added and deleted from objects, the set of identifiers bound by an object environment record may potentially change as a side-effect of any operation that adds or deletes properties. Any bindings that are created as a result of such a side-effect are considered to be a mutable binding even if the Writable attribute of the corresponding property has the value false. Immutable bindings do not exist for object environment records.

The behaviour of the concrete specification methods for Object Environment Records are defined by the following algorithms.

10.2.1.2.1 HasBinding(N)
The concrete Environment Record method HasBinding for object environment records determines if its associated binding object has a property whose name is the value of the argument $N$.
1. Let $envRec$ be the object environment record for which the method was invoked.
2. Let $bindings$ be the binding object for $envRec$.
3. Return the result of calling the [[HasProperty]] method of $bindings$, passing $N$ as the property name.

10.2.1.2.2 CreateMutableBinding (N)
The concrete Environment Record method CreateMutableBinding for object environment records creates a property whose name is the string value $N$ in the environment record and initializes it to the value undefined. A property named $N$ must not already exist in the binding object.
1. Let $envRec$ be the declarative environment record for which the method was invoked.
2. Let $bindings$ be the binding object for $envRec$.
3. Assert: The result of calling the [[HasProperty]] method of $bindings$, passing $N$ as the property name is false.
4. Call the [[Put]] method of $bindings$, passing $N$ and undefined for the arguments.

10.2.1.2.3 SetMutableBinding (N,V,S)
The concrete Environment Record method SetMutableValue for object environment records attempts to set the value of the environment record’s associated binding object’s property whose name is the value of the argument $N$ to the value of argument $V$. A property named $N$ should already exist but if it
does not or is not currently writable, error handling is determined by the value of the Boolean argument S.

1. Let envRec be the object environment record for which the method was invoked.
2. Let bindings be the binding object for envRec.
3. Call the [[ThrowingPut]] method of bindings with arguments N, V, and S.

10.2.1.2.4 GetBindingValue(N,S)

The concrete Environment Record method GetBindingValue for object environment records returns the value of it’s associated binding object’s property whose name is the string value of the argument identifier N. The property should already exist but if it does not the result depends upon the value of S argument:

1. Let envRec be the object environment record for which the method was invoked.
2. Let bindings be the binding object for envRec.
3. Let value be the result of calling the [[HasProperty]] method of bindings, passing N as the property name.
4. If value is false, then
   a. If S is false, return the value undefined, otherwise throw a ReferenceError exception.
5. Return the result of calling the [[Get]] method of bindings, passing N for the argument.

10.2.2 Lexical Environment Operations

The following abstract operations are used in this specification to operate upon lexical environments:

10.2.2.1 GetIdentifierReference (lex, name, strict)

The abstract operation GetIdentifierReference is called with a Lexical Environment lex, an identifier string name, and a boolean flag strict. The value of lex may be null. When called, the following steps are performed:

1. If lex is the value null, then
   a. Return a value of type Reference whose base value is null, whose referenced name is name, and whose strict mode flag is strict.
2. Let envRec be lex’s environment record.
3. Let exists be the result of calling the HasBinding(N) concrete method of envRec passing name as the argument N.
4. If exists is true, then
   a. Return a value of type Reference whose base value is envRec, whose referenced name is name, and whose strict mode flag is strict.
5. Else
   a. Let outer be the value of lex’s outer environment reference.
   b. Return the result of calling GetIdentifierReference passing outer, name, and strict as arguments.

10.2.2.2 NewDeclarativeEnvironmentRecord(E)

When the abstract operation NewDeclarativeEnvironmentRecord is called with either a Lexical Environment or null as argument E the following steps are performed:

1. Let env be a new Lexical Environment.
2. Let envRec be a new DeclarativeEnvironmentRecord containing no bindings.
3. Set env’s environment record to be envRec.
4. Set the outer lexical environment reference of env to E.
5. Return env.

10.2.2.3 NewObjectEnvironmentRecord(O, E)

When the abstract operation NewObjectEnvironmentRecord is called with an Object O and a Lexical Environment E (or null) as arguments, the following steps are performed:

1. Let env be a new Lexical Environment.
2. Let envRec be a new ObjectEnvironmentRecord containing using O as the binding object.
3. Set env’s environment record to be envRec.
4. Set the outer lexical environment reference of env to E.
5. Return env.
10.2.3 The Global Environment

The global environment is a unique Lexical Environment which is created before any ECMAScript code is executed. The global environment’s Environment Record is an object environment record whose binding object is the global object (15.1). The global environment’s outer environment reference is null.

As ECMAScript code is executed, additional properties may be added to the global object and the initial properties may be modified.

10.3 Execution Contexts

When control is transferred to ECMAScript executable code, control is entering an execution context. Active execution contexts logically form a stack. The top execution context on this logical stack is the running execution context. A new execution context is created whenever control is transferred from the executable code associated with the currently running execution context to executable code that is not associated with that execution context. The newly created execution context is pushed onto the stack and becomes the running execution context.

An execution context contains whatever state is necessary to track the execution progress of its associated code. In addition, each execution context has the following state components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>LexicalEnvironment</td>
<td>Identifies the Lexical Environment used to resolve identifier references made by code within this execution context.</td>
</tr>
<tr>
<td>VariableEnvironment</td>
<td>Identifies the Lexical Environment whose environment record holds bindings created by VariableStatements and FunctionDeclarations within this execution context.</td>
</tr>
<tr>
<td>ThisBinding</td>
<td>The value associated with the this keyword within ECMAScript code associated with this execution context.</td>
</tr>
</tbody>
</table>

The LexicalEnvironment and VariableEnvironment components of an execution context are always Lexical Environments. When an execution context is created its LexicalEnvironment and VariableEnvironment components initially have the same value. The value of the VariableEnvironment component never changes while the value of the LexicalEnvironment component may change during execution of code within an execution context.

In most situations only the running execution context (the top of the execution context stack) is directly manipulated by algorithms within this specification. Hence when the terms “LexicalEnvironment”, “VariableEnvironment” and “ThisBinding” are used without qualification they are in reference to those components of the running execution context.

An execution context is purely a specification mechanism and need not correspond to any particular artefact of an ECMAScript implementation. It is impossible for an ECMAScript program to access an execution context.

10.3.1 Identifier Resolution

Identifier resolution is the process of determining the binding of an Identifier using the LexicalEnvironment of the running execution context. During execution of ECMAScript code, the syntactic production PrimaryExpression : Identifier is evaluated using the following algorithm:

1. Let env be the running execution context’s LexicalEnvironment.
2. If the syntactic production that is being evaluated is contained in a strict mode code, then let strict be true else let strict be false.
3. Return the result of calling GetIdentifierReference function passing env, Identifier, and strict as arguments.

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

10.4 Establishing An Execution Context

Evaluation of global code or code using the eval function (15.1.2.1) establishes and enters a new execution context. Every invocation of a ECMAScript code function (13.2.1) also establishes and enters a new
execution context, even if a function is calling itself recursively. Every return exits an execution context. A thrown exception may also exit one or more execution contexts.

When control enters an execution context, the execution context’s ThisBinding is set, its VariableEnvironment and initial LexicalEnvironment are defined, and declaration binding instantiation is performed. The exact manner in which these actions occur depend on the type of code being entered.

10.4.1 Global Code
The following steps are performed when control enters the execution context for global code:

1. Initialize the execution context using the global code as described in 10.4.1.1.
2. Perform Declaration Binding Instantiation as described in 10.6 using the global code.

10.4.1.1 Initial Global Execution Context
The following steps are performed to initialize an execution context for ECMAScript code $C$:

1. Set the VariableEnvironment to the Global Environment.
2. Set the LexicalEnvironment to the Global Environment.
3. If $C$ is strict code, set the ThisBinding to $\text{undefined}$, otherwise set the ThisBinding to the global object.

10.4.2 Eval Code
The following sets are performed when control enters the execution context for eval code:

1. If there is no calling context or if the eval code is not being evaluated by a direct call (15.1.2.1) then,
   a. Initialize the execution context as if it was a global execution context using the eval code as $C$ as described in 10.4.1.1.
2. Else,
   a. Set the ThisBinding to the same value as the ThisBinding of the calling execution context.
   b. Set the LexicalEnvironment to the LexicalEnvironment of the calling execution context.
   c. Set the VariableEnvironment to the same value as the VariableEnvironment of the calling execution context.
3. If the eval code is strict code, then
   a. Let $\text{strictVarEnv}$ be the result of calling NewDeclarativeEnvironmentRecord(E) passing the LexicalEnvironment as the argument.
   b. Set the VariableEnvironment to $\text{strictVarEnv}$.
4. Perform Declaration Binding Instantiation as described in 10.6 using the eval code.

10.4.2.1 Strict Mode Restrictions
The eval code cannot instantiate variable or function bindings in the variable environment of the calling context that invoked the eval if either the code of the calling context or the eval code is strict code. Instead such bindings are instantiated in a new VariableEnvironment that is only accessible to the eval code.

10.4.3 Function Code
The following sets are performed when control enters the execution context for function code contained in function object $F$, a caller provided $\text{thisArg}$, and a caller provided $\text{argumentsList}$:

1. If the function code is strict code, set the ThisBinding to $\text{thisArg}$.
2. Else if $\text{thisArg}$ is null or $\text{undefined}$, set the ThisBinding to the global object.
3. Else if $\text{thisArg}$ is not an Object, set the ThisBinding to ToObject($\text{thisArg}$).
4. Else set the ThisBinding to $\text{thisArg}$.
5. Let $\text{localEnv}$ be the result of calling NewDeclarativeEnvironmentRecord(E) passing the value of the $\text{[[Scope]]}$ property of $F$ as the argument.
6. Set the LexicalEnvironment to $\text{localEnv}$.
7. Set the VariableEnvironment to $\text{localEnv}$.
8. Let $\text{code}$ be the value of $F$’s $\text{[[Code]]}$ internal property.
9. Perform Declaration Binding Instantiation using the function code $\text{code}$ and $\text{argumentsList}$ as described in 10.6.
10.5 Arguments Object

When control enters an execution context for function code, an arguments object is created.

The arguments object is created by calling the abstract operation CreateArgumentsObject with arguments `func`, the function object whose code is to be evaluated, `names` a List containing the formal parameter names, `args` the actual arguments pass to the `[[Call]]` method, `env` the variable environment for the function code, and `strict` a Boolean that indicates whether or not the function code is strict code. When CreateArgumentsObject is called the following steps are performed:

If `strict` is `true`, perform the following steps:

1. Let `len` be the number of elements in `args`.
2. Let `obj` be the result of creating a new object as if by the expression `new Array(len)` where `Array` is the standard built-in constructor with that name and `len` is the numeric value of `len`.
3. Let `indx = 0`.
4. Repeat while `indx < len`.
   a. Let `val` be the the element of `args` at 0-originated list position `indx`.
   b. Call the `[[Put]]` method of `obj` passing `indx` and `val` as arguments.
   c. Let `indx = indx + 1`.

5. Let `f` be a function which when evaluated throws a `TypeError` exception and performs no other actions.
6. Call the `[[DefineOwnProperty]]` method on `obj` passing `callee`, the property descriptor `[[Getter]]: f`, `[[Enumerable]]: false`, `[[Configurable]]: false`, and `true` as arguments.
7. Call the `[[DefineOwnProperty]]` method on `obj` passing `caller`, the property descriptor `[[Getter]]: f`, `[[Enumerable]]: false`, `[[Configurable]]: false`, and `true` as arguments.
8. Return `obj`.

If `strict` is `false`, perform the following steps:

1. Let `len` be the number of elements in `args`.
2. Let `obj` be the result of creating a new ECMAScript object.
3. Set the `[[Class]]` property of `obj` to "Object".
4. Set the `[[Prototype]]` property of `obj` to the standard built-in Object constructor (Section 15.2.3).
5. Set the `[[Prototype]]` property of `obj` to the standard built-in Array prototype object (Section 15.4.4).
6. Call the `[[DefineOwnProperty]]` method on `obj` passing "length", the property descriptor `[[Value]]: len`, `[[Enumerable]]: false`, `[[Configurable]]: true`, and `false` as arguments.
7. Let `indx = 0`.
8. Repeat while `indx < len`.
   a. If `indx` is less than the number of elements in `names`, then
      i. Let `name` be the the element of `names` at 0-originated list position `indx`.
      ii. Let `g` be the result of calling the MakeArgGetter function with arguments `name` and `env`.
      iii. Let `p` be the result of calling the MakeArgSetter function with arguments `name` and `env`.
      iv. Call the `[[DefineOwnProperty]]` method on `obj` passing `ToString(indx)`, the property descriptor `[[Getter]]: p`, `[[Setter]]: g`, `[[Enumerable]]: false`, `[[Configurable]]: true`, and `false` as arguments.
   b. Else, there are fewer arg names than actual arguments so
      i. Let `val` be the the element of `args` at 0-originated list position `indx`.
      ii. Call the `[[Put]]` method of `obj` passing `indx` and `val` as arguments.
   c. Let `indx = indx + 1`.
9. Call the `[[DefineOwnProperty]]` method on `obj` passing "callee", the property descriptor `[[Value]]: func`, `[[Enumerable]]: false`, `[[Configurable]]: true`, and `false` as arguments.
10. Return `obj`.

The function `MakeArgGetter` called with string `name` and environment record `env` creates a function object that when executes returns the value bound for `name` in `env`. It performs the following steps:

1. Let body be the result of concatenating the strings "return", `name`, and ";"
2. Create a function object as described in 13.2 using no `FormalParameterList`, body for `FunctionBody`, `env` as `Scope`, and the empty string as `Name`.

15 December 2008
3. Return Result(2)

The function MakeArgSetter called with string name and environment record env creates a function object that when executes returns the value bound for name in env. It performs the following steps:
1. Let param be the string name concatenated with the string ", arg"
2. Let body be the string "<name> = <param>;" with <name> replaced by the value of name and <param> replaced by the value of param.
3. Create a function object as described in 13.2 using a List containing the single string param as FormalParameterList, body for FunctionBody, env as Scope, and the empty string as Name.
4. Return Result(3).

10.5.1 Strict Mode Restrictions

If a arguments object is created, a callee property is not created.

The arguments object does not share properties with the activation object. Changing the value of a arguments object property does not change the value of the corresponding activation object property and vice versa.

10.6 Declaration Binding Instantiation

Every execution context has associated with a VariableEnvironment. Variables and functions declared in ECMAScript code evaluated in an execution context are added as bindings in that VariableEnvironment’s Environment Record. For function code, parameters are also added as bindings to that Environment Record.

Which Environment Record is used to bind declaration and its kind depends upon the type of ECMAScript code executed by the execution context, but the remainder of the behaviour is generic. On entering an execution context, bindings are created in the VariableEnvironment environment record as follows using the called provided code and (if it is function code) a function object func and argument list args:
1. Let env be the environment record component of the running execution context’s VariableEnvironment.
2. If code is strict mode code, then let strict be true else let strict be false.
3. If code is eval code, let eval be true, otherwise let eval be false.
4. If code is function code, then
   a. Let names be the value of func’s [[FormalParameters]] internal property.
   b. Let argCount be the number of elements in args.
   c. Let func be the function object that is the this value of the [[Call]] internal method that is is executing the function code.
   d. Let n be the number 0.
   e. For each string argName in names, in list order do
      i. Let n be the current value of n plus 1.
      ii. If n is greater than the number of elements in args, let v be undefined otherwise let v be the value of the n\text{th} element of args.
      iii. Let argAlreadyDeclared be the result of calling env’s HasBinding(N) concrete method passing argName as the argument.
      iv. If argAlreadyDeclared is false, call env’s CreateMutableBind(N) concrete method passing argName as the argument.
      v. Call env’s SetMutableBinding(N,V,S) concrete method passing argName, v, and strict as the arguments.
5. For each FunctionDeclaration f in the execution context’s code, in source text order do
   a. Let fn be the Identifier in FunctionDeclaration f.
   b. Let fo be the result of evaluating FunctionDeclaration for f as described in 13.
   c. Let funcAlreadyDeclared be the result of calling env’s HasBinding(N) concrete method passing fn as the argument.
   d. If funcAlreadyDeclared is false, call env’s CreateMutableBind(N) concrete method passing fn as the argument.
   e. Else if strict is true and eval is true throw an EvalError exception.
   f. Call env’s SetMutableBinding(N,V,S) concrete method passing fn, fo, and strict as the arguments.
6. For each VariableDeclaration and VariableDeclarationNoIn d in the execution context’s code, in source text order do
a. Let \( dn \) be the Identifier in \( d \).
b. Let \( \text{varAlreadyDeclared} \) be the result of calling \( \text{env}'s \) `HasBinding(N)` concrete method passing \( dn \) as the argument.
c. If \( \text{varAlreadyDeclared} \) is \( \text{false} \), then
   i. Call \( \text{env}'s \) `CreateMutableBind(N)` concrete method passing \( dn \) as the argument.
   ii. Call \( \text{env}'s \) `SetMutableBinding(N,V,S)` concrete method passing \( dn, \text{undefined} \), and
      \( \text{strict} \) as the arguments.
d. Else if \( \text{strict} \) is \( \text{true} \) and \( \text{eval} \) is \( \text{true} \) throw an \text{EvalError} exception.

7. Let \( \text{argumentsAlreadyDeclared} \) be the result of calling \( \text{env}'s \) `HasBinding(N)` concrete method passing
   "arguments" as the argument

8. If the code is function code and \( \text{argumentsAlreadyDeclared} \) is \( \text{false} \), then
   a. Let \( \text{argsObj} \) be the result of calling the abstract operation CreateArgumentsObject passing
      \( \text{func}, \text{names}, \text{args}, \text{env} \) and \( \text{strict} \) as arguments.
   b. If \( \text{strict} \) is \( \text{true} \), then
      i. Call \( \text{env}'s \) `CreateImmutableBinding(N)` concrete method passing the string
         "arguments" as the argument.
      ii. Call \( \text{env}'s \) `InitializeImmutableBind(N,V)` concrete method passing "arguments" and
          \( \text{argsObj} \) as arguments.
   c. Else,
      i. Call \( \text{env}'s \) `CreateMutableBind(N,D)` concrete method passing the string
         "arguments" and \( \text{false} \) as the arguments.
      ii. Call \( \text{env}'s \) `SetMutableBind(N,V,S)` concrete method passing "arguments", \( \text{argsObj} \),
          and \( \text{strict} \) as arguments.

11 Expressions

11.1 Primary Expressions

Syntax

```
PrimaryExpression :
  this
  Identifier
  Literal
  ArrayLiteral
  ObjectLiteral
  ( Expression )
```

11.1.1 The this Keyword

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

11.1.2 Identifier Reference

An `Identifier` is evaluated using the scoping rules stated in 10.3.1. The result of evaluating an `Identifier` is always a value of type `Reference`.

11.1.3 Literal Reference

A `Literal` is evaluated as described in 7.8.

11.1.4 Array Initialiser

An array initialiser is an expression describing the initialisation of an Array object, written in a form of a literal. It is a list of zero or more expressions, each of which represents an array element, enclosed in square brackets. The elements need not be literals; they are evaluated each time the array initialiser is evaluated.

Array elements may be elided at the beginning, middle or end of the element list. Whenever a comma in the element list is not preceded by an `AssignmentExpression` (i.e., a comma at the beginning or after another comma), the missing array element contributes to the length of the Array and increases the index of subsequent elements. Elided array elements are not defined. If an element is elided at the end of an array, that element does not contribute to the length of the Array.
ArrayLiteral : [ Elisionopt ]
  [ ElementList ]
  [ ElementList , Elisionopt ]

ElementList :
 Elisionopt AssignmentExpression
 ElementList , Elisionopt AssignmentExpression

Elision :
 ,

Semantics
The production `ArrayLiteral : [ Elisionopt ]` is evaluated as follows:
1. Let `array` be the result of creating a new object as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.
2. Let `pad` be the result of evaluating `Elision`; if not present, use the numeric value zero.
3. Call the [[Put]] internal method of `array` with arguments “length” and `pad`.
4. Return `array`.

The production `ArrayLiteral : [ ElementList ]` is evaluated as follows:
1. Return the result of evaluating `ElementList`.

The production `ArrayLiteral : [ ElementList , Elisionopt ]` is evaluated as follows:
1. Let `array` be the result of evaluating `ElementList`.
2. Let `pad` be the result of evaluating `Elision`; if not present, use the numeric value zero.
3. Let `len` be the result of calling the [[Get]] internal method of `array` with argument “length”.
4. Call the [[Put]] internal method of `array` with arguments “length” and `(pad+len)`.
5. Return `array`.

The production `ElementList : Elisionopt AssignmentExpression` is evaluated as follows:
1. Let `array` be the result of creating a new object as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.
2. Let `firstIndex` be the result of evaluating `Elision`; if not present, use the numeric value zero.
3. Let `initValue` be the result of evaluating `AssignmentExpression`.
4. Let `initResult` be the result of evaluating `AssignmentExpression`.
5. Call the [[Put]] internal method of `array` with arguments `firstIndex` and `initValue`.
6. Return `Result(1)`.

The production `ElementList : ElementList , Elisionopt AssignmentExpression` is evaluated as follows:
1. Let `array` be the result of evaluating `ElementList`.
2. Let `pad` be the result of evaluating `Elision`; if not present, use the numeric value zero.
3. Let `initResult` be the result of evaluating `AssignmentExpression`.
4. Let `initValue` be `GetValue(initResult)`. 
5. Call the [[Put]] internal method of `array` with arguments “length” and `pad+len`.
6. Call the [[Put]] internal method of `array` with arguments “length” and `initValue`.
7. Return `array`.

The production `Elision : ,` is evaluated as follows:
1. Return the numeric value 1.

The production `Elision : Elision ,` is evaluated as follows:
1. Let `preceding` be the result of evaluating `Elision`.
2. Return `preceding+1`.

15 December 2008
NOTE:
The use of \[\text{Put}\] rather than \[\text{ThrowingPut}\] in this section is intentional as there are no situations where these \[\text{Put}\] operations may fail.

### 11.1.5 Object Initialiser

An object initialiser is an expression describing the initialisation of an Object, written in a form resembling a literal. It is a list of zero or more pairs of property names and associated values, enclosed in curly braces. The values need not be literals; they are evaluated each time the object initialiser is evaluated.

#### Syntax

- **ObjectLiteral**: 
  - \{ \}
  - \{ PropertyNameAndValueList \}
  - \{ PropertyNameAndValueList , \}

- **PropertyNameAndValueList**: 
  - PropertyAssignment
  - PropertyNameAndValueList , PropertyAssignment

- **PropertyAssignment**: 
  - PropertyName : AssignmentExpression
  - `getProperty_Name() { FunctionBody }`

- **PropertyName**: 
  - Identifier
  - StringLiteral
  - NumericLiteral

- **PropertySetParameterList**: 
  - Identifier

#### Semantics

The production `ObjectLiteral : \{ \}` is evaluated as follows:

1. Create a new object as if by the expression `\new Object()` where `Object` is the standard built-in constructor with that name.
2. Return Result(1).

The productions `ObjectLiteral : \{ PropertyNameAndValueList \}` and `ObjectLiteral : \{ PropertyNameAndValueList , \}` are evaluated as follows:

1. Return the result of evaluating `PropertyNameAndValueList`.

The production `PropertyNameAndValueList : PropertyAssignment` is evaluated as follows:

1. 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.
2. Let `propId` be the result of evaluating `PropertyAssignment`.
3. Call the `\[\text{DefineOwnProperty}\]` internal method of `obj` with arguments `propId.name`, `propId.descriptor`, `false`.
4. Return `obj`.

The production `PropertyNameAndValueList : PropertyNameAndValueList , PropertyAssignment` is evaluated as follows:

15 December 2008
1. Let \( obj \) be the result of evaluating `PropertyNameAndValueList`.
2. Let \( propId \) be the result of evaluating `PropertyName`.
3. Let \( previous \) be the result of calling the `[[GetOwnProperty]]` internal method of \( obj \) with argument \( propId.name \).
4. If \( previous \) is not `undefined` then throw a SyntaxError exception if any of the following conditions are true:
   a. The production is contained in strict code and `IsAccessorDescriptor(previous)` is `true` and `IsPropertyDescriptor(propId.descriptor)` is `true`.
   b. `IsPropertyDescriptor(previous)` is `true` and `IsAccessorDescriptor(propId.descriptor)` is `true`.
   c. `IsAccessDescriptor(previous)` is `true` and `IsPropertyDescriptor(propId.descriptor)` is `true`.
   d. `IsPropertyDescriptor(previous)` is `true` and `IsPropertyDescriptor(propId.descriptor)` is `true` and either both `previous` and `propId.descriptor` have `[[Setter]]` fields or both `previous` and `propId.descriptor` have `[[Enumerable]]` fields.
5. Call the `[[DefineOwnProperty]]` internal method of \( obj \) with arguments `propId.name`, `previous`, \( propName \), \( exprValue \), `false`, `true`, and `false`.
6. Return \( gbi \).

If the above steps would throw a SyntaxError then an implementation must report the error immediately when scanning the program.

The production `PropertyAssignment : PropertyName : AssignmentExpression` is evaluated as follows:
1. Let \( propName \) be the result of evaluating `PropertyName`.
2. Let \( exprValue \) be the result of evaluating `AssignmentExpression`.
3. Let \( propValue \) be `GetValue(exprValue)`.
4. Let \( desc \) be the Property Descriptor `[[Value]]: propValue; [[Writable]]: true; [[Enumerable]]: true; [[Configurable]]: true`.
5. Return Property Identifier(`propName, desc`).

The production `PropertyAssignment : get PropertyName ( ) { FunctionBody }` is evaluated as follows:
1. Let \( propName \) be the result of evaluating `PropertyName`.
2. Let \( closure \) be the result of creating a new Function object as specified in 13.2 with an empty parameter list and body specified by `FunctionBody`. Pass in the Lexical Environment of the running execution context as the `Scope`, and `propName` as the `Name`.
3. Let \( desc \) be the Property Descriptor `[[Property]]: closure, [[Enumerable]]: true, [[Configurable]]: true`.
4. Return Property Identifier(`propName, desc`).

The production `PropertyAssignment : set PropertyName ( PropertySetParameterList ) { FunctionBody }` is evaluated as follows:
1. Let \( propName \) be the result of evaluating `PropertyName`.
2. Let \( closure \) be the result of creating a new Function object as specified in 13.2 with parameters specified by `PropertySetParameterList` and body specified by `FunctionBody`. Pass in the Lexical Environment of the running execution context as the `Scope`, and `propName` as the `Name`.
3. Let \( desc \) be the Property Descriptor `[[Setter]]: closure, [[Enumerable]]: true, [[Configurable]]: true`.
4. Return Property Identifier(`propName, desc`).

The production `PropertyName : IdentifierName` is evaluated as follows:
1. Return the string value containing the same sequence of characters as the `IdentifierName`.

The production `PropertyName : StringLiteral` is evaluated as follows:
1. Return the value of the `StringLiteral`.

The production `PropertyName : NumericLiteral` is evaluated as follows:
1. Let \( nb \) be the result of forming the value of the `NumericLiteral`.
2. Return `ToString(nb)`.
11.1.6 The Grouping Operator

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

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

NOTE
This algorithm does not apply GetValue to Result(1). The principal motivation for this is so that
operators such as delete and typeof may be applied to parenthesised expressions.

11.2 Left-Hand-Side Expressions

Syntax

MemberExpression :
  PrimaryExpression
  FunctionExpression
  MemberExpression [ Expression ]
  MemberExpression . Identifier
  new MemberExpression Arguments

NewExpression :
  MemberExpression
  new NewExpression

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

Arguments :
  ( )
  ( ArgumentList )

ArgumentList :
  AssignmentExpression
  ArgumentList , AssignmentExpression

LeftHandSideExpression :
  NewExpression
  CallExpression

11.2.1 Property Accessors

Properties are accessed by name, using either the dot notation:

  MemberExpression . Identifier
  CallExpression . Identifier

or the bracket notation:

  MemberExpression [ Expression ]
  CallExpression [ Expression ]

The dot notation is explained by the following syntactic conversion:

  MemberExpression . Identifier

is identical in its behaviour to
MemberExpression [ <identifier-name-string> ]

and similarly

CallExpression . IdentifierName

is identical in its behaviour to

CallExpression [ <identifier-name-string> ]

where <identifier-name-string> is a string literal containing the same sequence of characters after processing of Unicode escape sequences as the IdentifierName.

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

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

The production CallExpression : CallExpression [ Expression ] is evaluated in exactly the same manner, except that the contained CallExpression is evaluated in step 1.

11.2.2 The new Operator

The production NewExpression : new NewExpression is evaluated as follows:

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

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

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

11.2.3 Function Calls

The production CallExpression : MemberExpression Arguments is evaluated as follows:

1. Evaluate MemberExpression.
2. Call GetValue(Result(1)).
3. Evaluate Arguments, producing an internal list of argument values (see 11.2.4).
4. If Type(Result(2)) is not Object, throw a TypeError exception.
5. If IsCallable(Result(2)) is false, throw a TypeError exception.
6. If Type(Result(1)) is Reference, and IsPropertyReference(Result(1)) is true, Result(6) is GetBase(Result(1)). Otherwise, Result(6) is null.
7. Call the [[Call]] method on Result(2), providing Result(2) as the this value and providing the list Result(3) as the argument values.
8. Return Result(7).
11.2.4 Argument Lists

The evaluation of an argument list produces an internal list of values (see 8.8).

The production `Arguments : ( )` is evaluated as follows:
1. Return an empty internal list of values.

The production `Arguments : ( ArgumentList )` is evaluated as follows:
1. Evaluate `ArgumentList`.
2. Return Result(1).

The production `ArgumentList : AssignmentExpression` is evaluated as follows:
1. Evaluate `AssignmentExpression`.
2. Call GetValue(Result(1)).
3. Return an internal list whose sole item is Result(2).

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

11.2.5 Function Expressions

The production `MemberExpression : FunctionExpression` is evaluated as follows:
1. Evaluate `FunctionExpression`.
2. Return Result(1).

11.3 Postfix Expressions

Syntax

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

11.3.1 Postfix Increment Operator

The production `PostfixExpression : LeftHandSideExpression[no LineTerminator here] ++` is evaluated as follows:
1. Evaluate `LeftHandSideExpression`.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Add the value 1 to Result(3), using the same rules as for the `+` operator (see 11.6.3).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(3).
11.3.2 Postfix Decrement Operator

The production PostfixExpression : LeftHandSideExpression [no LineTerminator here] -- is evaluated as follows:
1. Evaluate LeftHandSideExpression.
2. Call GetValue(Result(1)).
3. Call ToNumber(Result(2)).
4. Subtract the value 1 from Result(3), using the same rules as for the – operator (11.6.3).
5. Call PutValue(Result(1), Result(4)).
6. Return Result(3).

11.4 Unary Operators

Syntax

UnaryExpression : PostfixExpression
delete UnaryExpression
void UnaryExpression
typeof UnaryExpression
++ UnaryExpression
-- UnaryExpression
* UnaryExpression
~ UnaryExpression
! UnaryExpression

11.4.1 The delete Operator

The production UnaryExpression : delete UnaryExpression is evaluated as follows:
1. Let ref be the result of evaluating UnaryExpression.
2. If Type(ref) is not Reference, return true.
3. If UnresolvableReference(ref) return true.
4. If IsPropertyReference(ref) is true, then
   a. Return the result of calling the [[Delete]] internal method on GetBase(ref) providing
   GetReferencedName(ref) and IsStrictReference(ref) as the arguments.
5. Else, ref is a Reference to an Environment Record binding, so
   a. If IsStrictReference(ref) is true throw a ReferenceError exception.
   b. If GetBase(ref) is a declarative environment record, return false.
   c. Let obj be the binding object of the object environment record that is the value of
      GetBase(ref).
   d. Return the result of calling the [[Delete]] internal method on obj, providing
      GetReferencedName(ref) and false as the arguments.

NOTE

When a delete operator occurs within strict mode code, a ReferenceError exception is thrown if its
UnaryExpression is a direct reference to a variable, function argument, or function name. In addition, if
the property to be deleted has the attribute { [[Configurable]]: false }, a TypeError exception is thrown.

11.4.2 The void Operator

The production UnaryExpression : void UnaryExpression is evaluated as follows:
1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Return undefined.

11.4.3 The typeof Operator

The production UnaryExpression : typeof UnaryExpression is evaluated as follows:
1. Evaluate UnaryExpression.
2. If Type(Result(1)) is not Reference, go to step 4.
3. If `IsUnresolvableReference(Result(1))` is `true`, return "undefined".
4. Call `GetValue(Result(1))`.
5. Return a string determined by `Type(Result(4))` according to the following table:

<table>
<thead>
<tr>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>&quot;undefined&quot;</td>
</tr>
<tr>
<td>Null</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Boolean</td>
<td>&quot;boolean&quot;</td>
</tr>
<tr>
<td>Number</td>
<td>&quot;number&quot;</td>
</tr>
<tr>
<td>String</td>
<td>&quot;string&quot;</td>
</tr>
<tr>
<td>Object (native and doesn’t implement <code>[[Call]]</code>)</td>
<td>&quot;object&quot;</td>
</tr>
<tr>
<td>Object (native or host and implements <code>[[Call]]</code>)</td>
<td>&quot;function&quot;</td>
</tr>
<tr>
<td>Object (host and doesn’t implement <code>[[Call]]</code>)</td>
<td>Implementation-dependent</td>
</tr>
</tbody>
</table>

11.4.4 Prefix Increment Operator

The production `UnaryExpression : ++ UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`.
2. Call `GetValue(Result(1))`.
3. Call `ToNumber(Result(2))`.
4. Add the value 1 to `Result(3)`, using the same rules as for the `+` operator (see 11.6.3).
5. Call `PutValue(Result(1), Result(4))`.
6. Return `Result(4)`.

11.4.5 Prefix Decrement Operator

The production `UnaryExpression : -- UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`.
2. Call `GetValue(Result(1))`.
3. Call `ToNumber(Result(2))`.
4. Subtract the value 1 from `Result(3)`, using the same rules as for the `–` operator (see 11.6.3).
5. Call `PutValue(Result(1), Result(4))`.
6. Return `Result(4)`.

11.4.6 Unary + Operator

The unary `+` operator converts its operand to Number type.

The production `UnaryExpression : + UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`.
2. Call `GetValue(Result(1))`.
3. Call `ToNumber(Result(2))`.
4. Return `Result(3)`.

11.4.7 Unary - Operator

The unary `-` operator converts its operand to Number type and then negates it. Note that negating `+0` produces `-0`, and negating `-0` produces `+0`.

The production `UnaryExpression : - UnaryExpression` is evaluated as follows:
1. Evaluate `UnaryExpression`.
2. Call `GetValue(Result(1))`.
3. Call `ToNumber(Result(2))`.

15 December 2008
4. If Result(3) is NaN, return NaN.
5. Negate Result(3); that is, compute a number with the same magnitude but opposite sign.
6. Return Result(5).

11.4.8 Bitwise NOT Operator (~)
The production UnaryExpression : ~ UnaryExpression is evaluated as follows:
1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToInt32(Result(2)).
4. Apply bitwise complement to Result(3). The result is a signed 32-bit integer.
5. Return Result(4).

11.4.9 Logical NOT Operator (!)
The production UnaryExpression : ! UnaryExpression is evaluated as follows:
1. Evaluate UnaryExpression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is true, return false.
5. Return true.

11.5 Multiplicative Operators

Syntax

MultiplicativeExpression :
UnaryExpression
MultiplicativeExpression * UnaryExpression
MultiplicativeExpression / UnaryExpression
MultiplicativeExpression % UnaryExpression

Semantics
The production MultiplicativeExpression : MultiplicativeExpression @ UnaryExpression, where @ stands for one of the operators in the above definitions, is evaluated as follows:
1. Evaluate MultiplicativeExpression.
2. Call GetValue(Result(1)).
3. Evaluate UnaryExpression.
4. Call GetValue(Result(3)).
5. Call ToNumber(Result(2)).
6. Call ToNumber(Result(4)).
7. Apply the specified operation (*, /, or %) to Result(5) and Result(6). See the notes below (11.5.1, 11.5.2, 11.5.3).
8. Return Result(7).

11.5.1 Applying the * Operator
The * operator performs multiplication, producing the product of its operands. Multiplication is commutative. Multiplication is not always associative in ECMAScript, because of finite precision.
The result of a floating-point multiplication is governed by the rules of IEEE 754 double-precision arithmetic:
If either operand is NaN, the result is NaN.
The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
Multiplication of an infinity by a zero results in NaN.
Multiplication of an infinity by an infinity results in an infinity. The sign is determined by the rule already stated above.
Multiplication of an infinity by a finite non-zero value results in a signed infinity. The sign is determined by the rule already stated above.
In the remaining cases, where neither an infinity or NaN is involved, the product is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the result is then a zero of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

11.5.2 Applying the / Operator

The / operator performs division, producing the quotient of its operands. The left operand is the dividend and the right operand is the divisor. ECMAScript does not perform integer division. The operands and result of all division operations are double-precision floating-point numbers. The result of division is determined by the specification of IEEE 754 arithmetic:

If either operand is NaN, the result is NaN.
The sign of the result is positive if both operands have the same sign, negative if the operands have different signs.
Division of an infinity by an infinity results in NaN.
Division of an infinity by a zero results in an infinity. The sign is determined by the rule already stated above.
Division of a finite value by a non-zero finite value results in a signed infinity. The sign is determined by the rule already stated above.
Division of a zero by a zero results in NaN; division of zero by any other finite value results in zero, with the sign determined by the rule already stated above.
Division of a non-zero finite value by a zero results in a signed infinity. The sign is determined by the rule already stated above.
In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the quotient is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the operation overflows; the result is then an infinity of appropriate sign. If the magnitude is too small to represent, the operation underflows and the result is a zero of the appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

11.5.3 Applying the % Operator

The % operator yields the remainder of its operands from an implied division; the left operand is the dividend and the right operand is the divisor.

NOTE
In C and C++, the remainder operator accepts only integral operands; in ECMAScript, it also accepts floating point operands.

The result of a floating-point remainder operation as computed by the % operator is not the same as the “remainder” operation defined by IEEE 754. The IEEE 754 “remainder” operation computes the remainder from a rounding division, not a truncating division, and so its behaviour is not analogous to that of the usual integer remainder operator. Instead the ECMAScript language defines % on floating-point operations to behave in a manner analogous to that of the Java integer remainder operator; this may be compared with the C library function fmod.

The result of a ECMAScript floating-point remainder operation is determined by the rules of IEEE arithmetic:

If either operand is NaN, the result is NaN.
The sign of the result equals the sign of the dividend.
If the dividend is an infinity, or the divisor is a zero, or both, the result is NaN.
If the dividend is finite and the divisor is an infinity, the result equals the dividend.
If the dividend is a zero and the divisor is finite, the result is the same as the dividend.
In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, the floating-point remainder r from a dividend n and a divisor d is defined by the mathematical relation \( r = n - (d \times q) \)
where \( q \) is an integer that is negative only if \( n/d \) is negative and positive only if \( n/d \) is positive, and whose magnitude is as large as possible without exceeding the magnitude of the true mathematical quotient of \( n \) and \( d \).

### 11.6 Additive Operators

#### Syntax

\[
\text{AdditiveExpression} : \ \\
\text{MultiplicativeExpression} \ \\
\text{AdditiveExpression} + \text{MultiplicativeExpression} \\
\text{AdditiveExpression} - \text{MultiplicativeExpression}
\]

#### 11.6.1 The Addition operator (\( + \))

The addition operator either performs string concatenation or numeric addition.

The production \( \text{AdditiveExpression} : \text{AdditiveExpression} + \text{MultiplicativeExpression} \) is evaluated as follows:

1. Evaluate \( \text{AdditiveExpression} \).
2. Call GetValue(Result(1)).
3. Evaluate \( \text{MultiplicativeExpression} \).
4. Call GetValue(Result(3)).
5. Call ToPrimitive(Result(2)).
6. Call ToPrimitive(Result(4)).
7. If Type(Result(5)) is String or Type(Result(6)) is String, then:
   a. Call ToString(Result(5)).
   b. Call ToString(Result(6)).
   c. Concatenate Result(7a) followed by Result(7b).
   d. Return Result(7c).
8. Call ToNumber(Result(5)).
9. Call ToNumber(Result(6)).
10. Apply the addition operation to Result(8) and Result(9). See the note below (11.6.3).
11. Return Result(10).

**NOTE**

No hint is provided in the calls to ToPrimitive in steps 5 and 6. All native ECMAScript objects except Date objects handle the absence of a hint as if the hint Number were given; Date objects handle the absence of a hint as if the hint String were given. Host objects may handle the absence of a hint in some other manner.

#### 11.6.2 The Subtraction Operator (\( - \))

The production \( \text{AdditiveExpression} : \text{AdditiveExpression} - \text{MultiplicativeExpression} \) is evaluated as follows:

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

#### 11.6.3 Applying the Additive Operators (\( +, - \)) to Numbers

The \( + \) operator performs addition when applied to two operands of numeric type, producing the sum of the operands. The \( - \) operator performs subtraction, producing the difference of two numeric operands.

Addition is a commutative operation, but not always associative.

The result of an addition is determined using the rules of IEEE 754 double-precision arithmetic:

If either operand is \( \text{NaN} \), the result is \( \text{NaN} \).
The sum of two infinities of opposite sign is \( \text{NaN} \).
The sum of two infinities of the same sign is the infinity of that sign.
The sum of an infinity and a finite value is equal to the infinite operand.
The sum of two negative zeros is \( -0 \). The sum of two positive zeros, or of two zeros of opposite sign, is \( +0 \).
The sum of a zero and a nonzero finite value is equal to the nonzero operand.
The sum of two nonzero finite values of the same magnitude and opposite sign is \( +0 \).
In the remaining cases, where neither an infinity, nor a zero, nor NaN is involved, and the operands have the same sign or have different magnitudes, the sum is computed and rounded to the nearest representable value using IEEE 754 round-to-nearest mode. If the magnitude is too large to represent, the operation overflows and the result is then an infinity of appropriate sign. The ECMAScript language requires support of gradual underflow as defined by IEEE 754.

The \(-\) operator performs subtraction when applied to two operands of numeric type, producing the difference of its operands; the left operand is the minuend and the right operand is the subtrahend. Given numeric operands \( a \) and \( b \), it is always the case that \( a - b \) produces the same result as \( a + (-b) \).

### 11.7 Bitwise Shift Operators

#### Syntax

\[
\text{ShiftExpression} : \\
\text{AdditiveExpression} \\
\text{ShiftExpression} \ll \text{AdditiveExpression} \\
\text{ShiftExpression} \gg \text{AdditiveExpression} \\
\text{ShiftExpression} \gg\gg \text{AdditiveExpression}
\]

#### 11.7.1 The Left Shift Operator ( \( \ll \) )

Performs a bitwise left shift operation on the left operand by the amount specified by the right operand.

The production \( \text{ShiftExpression} : \text{ShiftExpression} \ll \text{AdditiveExpression} \) is evaluated as follows:

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

#### 11.7.2 The Signed Right Shift Operator ( \( \gg \) )

Performs a sign-filling bitwise right shift operation on the left operand by the amount specified by the right operand.

The production \( \text{ShiftExpression} : \text{ShiftExpression} \gg \text{AdditiveExpression} \) is evaluated as follows:

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

Performs a zero-filling bitwise right shift operation on the left operand by the amount specified by the right operand.

The production `ShiftExpression : ShiftExpression >>> AdditiveExpression` is evaluated as follows:

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

11.8 Relational Operators

Syntax

RelationalExpression :
  ShiftExpression
  RelationalExpression < ShiftExpression
  RelationalExpression > ShiftExpression
  RelationalExpression <= ShiftExpression
  RelationalExpression >= ShiftExpression
  RelationalExpression instanceof ShiftExpression
  RelationalExpression in ShiftExpression

RelationalExpressionNoIn :
  ShiftExpression
  RelationalExpressionNoIn < ShiftExpression
  RelationalExpressionNoIn > ShiftExpression
  RelationalExpressionNoIn <= ShiftExpression
  RelationalExpressionNoIn >= ShiftExpression
  RelationalExpressionNoIn instanceof ShiftExpression

NOTE

The 'NoIn' variants are needed to avoid confusing the `in` operator in a relational expression with the `in` operator in a `for` statement.

Semantics

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

The `RelationalExpressionNoIn` productions are evaluated in the same manner as the `RelationalExpression` productions except that the contained `RelationalExpressionNoIn` is evaluated instead of the contained `RelationalExpression`.

11.8.1 The Less-than Operator ( < )

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

1. Evaluate `RelationalExpression`.
2. Call `GetValue(Result(1))`.
3. Evaluate `ShiftExpression`.
4. Call `GetValue(Result(3))`.
5. Perform the comparison `Result(2) < Result(4)`. (see 11.8.5)
6. If `Result(j)` is `undefined`, return false. Otherwise, return `Result(j)`.
11.8.2 The Greater-than Operator ( > )
The production \texttt{RelationalExpression : RelationalExpression \textgreater ShiftExpression} is evaluated as follows:

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) < Result(2) with \texttt{LeftFirst} equal to false. (see 11.8.5).
6. If Result(p) is \texttt{undefined}, return false. Otherwise, return Result(p).

11.8.3 The Less-than-or-equal Operator ( <= )
The production \texttt{RelationalExpression : RelationalExpression \leq ShiftExpression} is evaluated as follows:

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) < Result(2) with \texttt{LeftFirst} equal to false. (see 11.8.5).
6. If Result(p) is true or undefined, return false. Otherwise, return true.

11.8.4 The Greater-than-or-equal Operator ( >= )
The production \texttt{RelationalExpression : RelationalExpression \geq ShiftExpression} is evaluated as follows:

1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(2) < Result(4). (see 11.8.5).
6. If Result(p) is true or undefined, return false. Otherwise, return true.

11.8.5 The Abstract Relational Comparison Algorithm
The comparison \texttt{x \textless y}, where \texttt{x} and \texttt{y} are values, produces \texttt{true}, \texttt{false}, or \texttt{undefined} (which indicates that at least one operand is \texttt{NaN}). In addition to \texttt{x} and \texttt{y} the algorithm takes a boolean flag named \texttt{LeftFirst} as a parameter. The flag is used to control the order in which operations with potentially visible side-effects are performed upon \texttt{x} and \texttt{y}. It is necessary because ECMAScript specifies left to right evaluation of expressions. The default value of \texttt{LeftFirst} is \texttt{true} and indicates that the \texttt{x} parameter corresponds to an expression that occurs to the left of the \texttt{y} parameters corresponding expression. If \texttt{LeftFirst} is \texttt{false}, the reverse is the case and operations must be performed upon \texttt{y} before \texttt{x}. Such a comparison is performed as follows:

1. If the \texttt{LeftFirst} flag is \texttt{true}, then
   a. Let \texttt{px} be the result of calling \texttt{ToPrimitive(x, hint:Number)}.
   b. Let \texttt{py} be the result of calling \texttt{ToPrimitive(y, hint:Number)}.

2. Else the order of evaluation needs to be reversed to preserve left to right evaluation
   a. Let \texttt{py} be the result of calling \texttt{ToPrimitive(y, hint:Number)}.
   b. Let \texttt{px} be the result of calling \texttt{ToPrimitive(x, hint:Number)}.

3. If \texttt{Type(px)} is \texttt{String} or \texttt{Type(py)} is \texttt{String}, go to step 16. (Note that this step differs from step 7 in the algorithm for the addition operator \texttt{+} in using \texttt{and} instead of \texttt{or}.)

4. Let \texttt{nx} be the result of calling \texttt{ToNumber(px)}. Because of \texttt{px} and \texttt{py} are primitive values evaluation order is not important.

5. Let \texttt{ny} be the result of calling \texttt{ToNumber(py)}.

6. If \texttt{px} is \texttt{NaN}, return \texttt{undefined}.
7. If \texttt{py} is \texttt{NaN}, return \texttt{undefined}.

8. If \texttt{px} and \texttt{py} are the same number value, return \texttt{false}.
9. If \texttt{px} is \texttt{-0} and \texttt{py} is \texttt{-0}, return \texttt{false}.
10. If \texttt{px} is \texttt{-0} and \texttt{ny} is \texttt{+0}, return \texttt{false}.
11. If \texttt{nx} is \texttt{+0}, return \texttt{false}.
12. If \texttt{nx} is \texttt{-0}, return \texttt{true}.
13. If \texttt{nx} is \texttt{-0}, return \texttt{false}.
14. If \texttt{nx} is \texttt{-0}, return \texttt{true}.

15 December 2008
15. If the mathematical value of \( p_x \) is less than the mathematical value of \( q_y \)—note that these mathematical values are both finite and not both zero—return true. Otherwise, return false.

16. If \( p_x \) is a prefix of \( q_y \), return false. (A string value \( p \) is a prefix of string value \( q \) if \( q \) can be the result of concatenating \( p \) and some other string \( r \). Note that any string is a prefix of itself, because \( r \) may be the empty string.)

17. If \( p_x \) is a prefix of \( q_y \), return true.

18. Let \( k \) be the smallest nonnegative integer such that the character at position \( k \) within \( p_x \) is different from the character at position \( k \) within \( q_y \). (There must be such a \( k \), for neither string is a prefix of the other.)

19. Let \( n \) be the integer that is the code point value for the character at position \( k \) within \( p_x \).

20. Let \( m \) be the integer that is the code point value for the character at position \( k \) within \( q_y \).

21. If \( m < n \), return true. Otherwise, return false.

NOTE
The comparison of strings uses a simple lexicographic ordering on sequences of code point values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the Unicode specification. Therefore strings that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both strings are already in normalised form.

11.8.6 The instanceof operator
The production RelationalExpression: RelationalExpression instanceof ShiftExpression is evaluated as follows:
1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. If Result(4) is not an object, throw a TypeError exception.
6. If Result(4) does not have a [[HasInstance]] method, throw a TypeError exception.
7. Call the [[HasInstance]] method of Result(4) with parameter Result(2).
8. Return Result(7).

11.8.7 The in operator
The production RelationalExpression : RelationalExpression in ShiftExpression is evaluated as follows:
1. Evaluate RelationalExpression.
2. Call GetValue(Result(1)).
3. Evaluate ShiftExpression.
4. Call GetValue(Result(3)).
5. If Result(4) is not an object, throw a TypeError exception.
6. Call ToString(Result(2)).
7. Call the [[HasProperty]] method of Result(4) with parameter Result(6).
8. Return Result(7).

11.9 Equality Operators
Syntax
EqualityExpression :
RelationalExpression
EqualityExpression == RelationalExpression
EqualityExpression != RelationalExpression
EqualityExpression === RelationalExpression
EqualityExpression !== RelationalExpression
EqualityExpressionNoIn:
RelationalExpressionNoIn
EqualityExpressionNoIn == RelationalExpressionNoIn
EqualityExpressionNoIn != RelationalExpressionNoIn
EqualityExpressionNoIn !== RelationalExpressionNoIn
EqualityExpressionNoIn !=== RelationalExpressionNoIn

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

The EqualityExpressionNoIn productions are evaluated in the same manner as the EqualityExpression productions except that the contained EqualityExpressionNoIn and RelationalExpressionNoIn are evaluated instead of the contained EqualityExpression and RelationalExpression, respectively.

11.9.1 The Equals Operator (==)
The production EqualityExpression : EqualityExpression == RelationalExpression is evaluated as follows:
1. Evaluate EqualityExpression.
2. Call GetValue(Result(1)).
3. Evaluate RelationalExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) == Result(2). (see 11.9.3).
6. Return Result(5).

11.9.2 The Does-not-equals Operator (!=)
The production EqualityExpression : EqualityExpression != RelationalExpression is evaluated as follows:
1. Evaluate EqualityExpression.
2. Call GetValue(Result(1)).
3. Evaluate RelationalExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) == Result(2). (see 11.9.3).
6. If Result(5) is true, return false. Otherwise, return true.

11.9.3 The Abstract Equality Comparison Algorithm
The comparison \( x == y \), where \( x \) and \( y \) are values, produces true or false. Such a comparison is performed as follows:
1. If Type(\( x \)) is different from Type(\( y \)), go to step 14.
2. If Type(\( x \)) is Undefined, return true.
3. If Type(\( x \)) is Null, return true.
4. If Type(\( x \)) is not Number, go to step 11.
5. If \( x \) is NaN, return false.
6. If \( y \) is NaN, return false.
7. If \( x \) is the same number value as \( y \), return true.
8. If \( x \) is +0 and \( y \) is –0, return true.
9. If \( x \) is –0 and \( y \) is +0, return true.
10. Return false.
11. If Type(\( x \)) is String, then return true if \( x \) and \( y \) are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return false.
12. If Type(\( x \)) is Boolean, return true if \( x \) and \( y \) are both true or both false. Otherwise, return false.
13. Return true if \( x \) and \( y \) refer to the same object. Otherwise, return false.
14. If \( x \) is null and \( y \) is undefined, return true.
15. If \( x \) is undefined and \( y \) is null, return true.
16. If Type(\( x \)) is Number and Type(\( y \)) is String, return the result of the comparison \( x == \text{ToNumber}(y) \).

15 December 2008
17. If Type(x) is String and Type(y) is Number, return the result of the comparison ToNumber(x) == y.
18. If Type(x) is Boolean, return the result of the comparison ToNumber(x) == y.
19. If Type(y) is Boolean, return the result of the comparison x == ToNumber(y).
20. If Type(x) is either String or Number and Type(y) is Object, return the result of the comparison x == ToPrimitive(y).
21. If Type(x) is Object and Type(y) is either String or Number, return the result of the comparison ToPrimitive(x) == y.
22. Return false.

NOTE
Given the above definition of equality:
String comparison can be forced by: 
Numeric comparison can be forced by: 
Boolean comparison can be forced by:  !a == !b.

The equality operators maintain the following invariants:
A !== B is equivalent to ! (A == B).
A == B is equivalent to B == A, except in the order of evaluation of A and B.

The equality operator is not always transitive. For example, there might be two distinct String objects, each representing the same string value: each String object would be considered equal to the string value by the == operator, but the two String objects would not be equal to each other.

Comparison of strings uses a simple equality test on sequences of code point value values. There is no attempt to use the more complex, semantically oriented definitions of character or string equality and collating order defined in the Unicode 2.0 specification. Therefore strings that are canonically equal according to the Unicode standard could test as unequal. In effect this algorithm assumes that both strings are already in normalised form.

11.9.4 The Strict Equals Operator ( === )
The production EqualityExpression : EqualityExpression === RelationalExpression is evaluated as follows:
1. Evaluate EqualityExpression.
2. Call GetValue(Result(1)).
3. Evaluate RelationalExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) === Result(2). (See below.)
6. Return Result(5).

11.9.5 The Strict Does-not-equal Operator ( !== )
The production EqualityExpression : EqualityExpression !== RelationalExpression is evaluated as follows:
1. Evaluate EqualityExpression.
2. Call GetValue(Result(1)).
3. Evaluate RelationalExpression.
4. Call GetValue(Result(3)).
5. Perform the comparison Result(4) !== Result(2). (See below.)
6. If Result(5) is true, return false. Otherwise, return true.

11.9.6 The Strict Equality Comparison Algorithm
The comparison x === y, where x and y are values, produces true or false. Such a comparison is performed as follows:
1. If Type(x) is different from Type(y), return false.
2. If Type(x) is Undefined, return true.
3. If Type(x) is Null, return true.
4. If Type(x) is not Number, go to step 11.
5. If x is NaN, return false.
6. If y is NaN, return false.
7. If x is the same number value as y, return true.
8. If x is +0 and y is −0, return true.
9. If x is −0 and y is +0, return true.
10. Return false.
11. If Type(x) is String, then return true if x and y are exactly the same sequence of characters (same length and same characters in corresponding positions); otherwise, return false.
12. If Type(x) is Boolean, return true if x and y are both true or both false; otherwise, return false.
13. Return true if x and y refer to the same object. Otherwise, return false.

11.10 Binary Bitwise Operators

Syntax

BitwiseANDExpression :
  EqualityExpression
  BitwiseANDExpression & EqualityExpression

BitwiseXORExpression :
  BitwiseANDExpression
  BitwiseXORExpression ^ BitwiseANDExpression

BitwiseORExpression :
  BitwiseXORExpression
  BitwiseORExpression | BitwiseXORExpression

Semantics

The production A : A @ B, where @ is one of the bitwise operators in the productions above, is evaluated as follows:
1. Evaluate A.
2. Call GetValue(Result(1)).
3. Evaluate B.
4. Call GetValue(Result(3)).
5. Call ToInt32(Result(2)).
6. Call ToInt32(Result(4)).
7. Apply the bitwise operator @ to Result(5) and Result(6). The result is a signed 32 bit integer.
8. Return Result(7).

11.11 Binary Logical Operators

Syntax

LogicalANDExpression :
  BitwiseORExpression
  LogicalANDExpression & BitwiseORExpression
Semantics
The production LogicalANDExpression : LogicalANDExpression && BitwiseORExpression is evaluated as follows:
1. Evaluate LogicalANDExpression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is \texttt{false}, return Result(2).
5. Evaluate BitwiseORExpression.
6. Call GetValue(Result(5)).
7. Return Result(6).

The production LogicalORExpression : LogicalORExpression || LogicalANDExpression is evaluated as follows:
1. Evaluate LogicalORExpression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is \texttt{true}, return Result(2).
5. Evaluate LogicalANDExpression.
6. Call GetValue(Result(5)).
7. Return Result(6).

The LogicalANDExpressionNoIn and LogicalORExpressionNoIn productions are evaluated in the same manner as the LogicalANDExpression and LogicalORExpression productions except that the contained LogicalANDExpressionNoIn, BitwiseORExpressionNoIn and LogicalORExpressionNoIn are evaluated instead of the contained LogicalANDExpression, BitwiseORExpression and LogicalORExpression, respectively.

NOTE
The value produced by a \texttt{&&} or \texttt{||} operator is not necessarily of type Boolean. The value produced will always be the value of one of the two operand expressions.

11.12 Conditional Operator ( ? : )

Syntax
ConditionalExpression :
LogicalORExpression
LogicalORExpression \texttt{?} AssignmentExpression : AssignmentExpression

ConditionalExpressionNoIn :
LogicalORExpressionNoIn
LogicalORExpressionNoIn \texttt{?} AssignmentExpression : AssignmentExpressionNoIn

Semantics
The production ConditionalExpression : LogicalORExpression ? AssignmentExpression : AssignmentExpression is evaluated as follows:
1. Evaluate LogicalORExpression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, go to step 8.
5. Evaluate the first AssignmentExpression.
6. Call GetValue(Result(5)).
7. Return Result(6).
8. Evaluate the second AssignmentExpression.
9. Call GetValue(Result(8)).
10. Return Result(9).

The ConditionalExpressionNoIn production is evaluated in the same manner as the ConditionalExpression production except that the contained LogicalORExpressionNoIn, AssignmentExpression and AssignmentExpressionNoIn are evaluated instead of the contained LogicalORExpression, first AssignmentExpression and second AssignmentExpression, respectively.

NOTE
The grammar for a ConditionalExpression in ECMAScript is a little bit different from that in C and Java, which each allow the second subexpression to be an Expression but restrict the third expression to be a ConditionalExpression. The motivation for this difference in ECMAScript is to allow an assignment expression to be governed by either arm of a conditional and to eliminate the confusing and fairly useless case of a comma expression as the centre expression.

11.13 Assignment Operators

Syntax

AssignmentExpression : ConditionalExpression
LeftHandSideExpression AssignmentOperator AssignmentExpression

AssignmentExpressionNoIn : ConditionalExpressionNoIn
LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn

AssignmentOperator : one of
= *= /= %= += -= <<= >>= >>>= &= ^= |=

Semantics

The AssignmentExpressionNoIn productions are evaluated in the same manner as the AssignmentExpression productions except that the contained ConditionalExpressionNoIn and AssignmentExpressionNoIn are evaluated instead of the contained ConditionalExpression and AssignmentExpression, respectively.

11.13.1 Simple Assignment ( = )

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

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

NOTE
When a simple assignment occurs within strict mode code, its LeftHandSide must not evaluate to an unresolvable reference. If it does a ReferenceError exception is thrown upon assignment. The LeftHandSide also may not be a reference to a property with the attribute value {[[Writable]]: false} nor to a non-existent property of an object whose {[[Extensible]]} property has the value false. In these cases a TypeError exception is thrown.

15 December 2008
11.13.2 Compound Assignment ( \texttt{op} = )

The production \texttt{AssignmentExpression : LeftHandSideExpression \texttt{op} = AssignmentExpression}, where \texttt{op} represents one of the operators indicated above, is evaluated as follows:

1. Evaluate \texttt{LeftHandSideExpression}.
2. Call \texttt{GetValue(Result(1)).}
3. Evaluate \texttt{AssignmentExpression}.
4. Call \texttt{GetValue(Result(3)).}
5. Apply operator \texttt{op} to Result(2) and Result(4).
6. Call \texttt{PutValue(Result(1), Result(5)).}
7. Return Result(5).

11.13.2.1 Strict Mode Restrictions

The same restrictions apply as specified in 11.13.1.1.

11.14 Comma Operator ( , )

Syntax

\texttt{Expression : AssignmentExpression-expression, AssignmentExpression-expression}

\texttt{ExpressionNoIn : AssignmentExpressionNoIn-expression, AssignmentExpressionNoIn-expression}

Semantics

The production \texttt{Expression : Expression-expression, AssignmentExpression-expression} is evaluated as follows:

1. Evaluate \texttt{Expression}.
2. Call \texttt{GetValue(Result(1)).}
3. Evaluate \texttt{AssignmentExpression}.
4. Call \texttt{GetValue(Result(3)).}
5. Return Result(4).

The \texttt{ExpressionNoIn} production is evaluated in the same manner as the \texttt{Expression} production except that the contained \texttt{ExpressionNoIn} and \texttt{AssignmentExpressionNoIn} are evaluated instead of the contained \texttt{Expression} and \texttt{AssignmentExpression}, respectively.
12 Statements
Syntax
Statement : Block
            EmptyStatement
            ExpressionStatement
            IfStatement
            IterationStatement
            ContinueStatement
            BreakStatement
            ReturnStatement
            WithStatement
            LabelledStatement
            SwitchStatement
            ThrowStatement
            TryStatement
            DebuggerStatement

Semantics
A Statement can be part of a LabelledStatement, which itself can be part of a LabelledStatement, and so on.
The labels introduced this way are collectively referred to as the “current label set” when describing
the semantics of individual statements. A LabelledStatement has no semantic meaning other than the introduction
of a label to a label set. The label set of an IterationStatement or a SwitchStatement initially contains the
single element empty. The label set of any other statement is initially empty.

Note:
TBD: Implementations have been known to support FunctionDeclaration in a Statement; however there is no
uniform support. It is impossible to reconcile their differing semantics, and hence this specification excludes
their possibility.

12.1 Block
Syntax
Block : { StatementListopt }
StatementList : Statement
StatementList Statement

Semantics
The production Block : { } is evaluated as follows:
1. Return (normal, empty, empty).

The production Block : { StatementList } is evaluated as follows:
1. Evaluate StatementList.
2. Return Result(1).

The production StatementList : Statement is evaluated as follows:
1. Evaluate Statement.
2. If an exception was thrown, return (throw, V, empty) where V is the exception. (Execution now
proceeds as if no exception were thrown.)
3. Return Result(1).

The production StatementList : StatementList Statement is evaluated as follows:
1. Evaluate StatementList.
2. If Result(1) is an abrupt completion, return Result(1).
3. Evaluate Statement.
4. If an exception was thrown, return (throw, V, empty) where V is the exception. (Execution now proceeds as if no exception were thrown.)
5. If Result(3).value is empty, let V = Result(1).value, otherwise let V = Result(3).value.

12.2 Variable statement

Syntax

VariableStatement : var VariableDeclarationList ;

VariableDeclarationList : VariableDeclaration
VariableDeclarationList , VariableDeclaration

VariableDeclarationListNoIn : VariableDeclarationNoIn
VariableDeclarationListNoIn , VariableDeclarationNoIn

VariableDeclaration : Identifier Initialiser_opt

VariableDeclarationNoIn : Identifier InitialiserNoIn

Initialiser : = AssignmentExpression

InitialiserNoIn : = AssignmentExpressionNoIn

Description

A variable statement declares variables that are created as defined in section 10.6. Variables are initialised to undefined when created. A variable with an Initialiser is assigned the value of its AssignmentExpression when the VariableStatement is executed, not when the variable is created.

Semantics

The production VariableStatement : var VariableDeclarationList ; is evaluated as follows:

1. Evaluate VariableDeclarationList.
2. Return (normal, empty, empty).

The production VariableDeclarationList : VariableDeclaration is evaluated as follows:

1. Evaluate VariableDeclaration.

The production VariableDeclarationList : VariableDeclarationList , VariableDeclaration is evaluated as follows:

1. Evaluate VariableDeclarationList.
2. Evaluate VariableDeclaration.

The production VariableDeclaration : Identifier is evaluated as follows:

1. Return a string value containing the same sequence of characters as in the Identifier.

The production VariableDeclaration : Identifier Initialiser is evaluated as follows:

Deleted: If the

Deleted: occurs inside a FunctionDeclaration, the variables are defined with function-local scope in that function, as described in 10.1.3. Otherwise, they are defined with global scope (that is, they are created as members of the global object, as described in 10.1.3) using property attributes { DontDelete }. Variables are created when the execution scope is entered. A Block does not define a new execution scope. Only Program and FunctionDeclaration produce a new scope. Only Variables
1. If the VariableDeclaration occurs in strict mode code, let strict be true, otherwise let strict be false.
2. Let rhs be the result of evaluating Initialiser.
3. Let value be GetValue(rhs).
4. Call the SetMutableBinding(N, V, S) concrete method of the execution context’s VariableEnvironment passing the Identifier, value, and strict as arguments.
5. Return a string value containing the same sequence of characters as in the Identifier.

The production Initialiser : = AssignmentExpression is evaluated as follows:
1. Evaluate AssignmentExpression.
2. Return Result(1).

The VariableDeclarationListNoIn, VariableDeclarationNoIn and InitialiserNoIn productions are evaluated in the same manner as the VariableDeclarationList, VariableDeclaration and Initialiser productions except that the contained VariableDeclarationListNoIn, VariableDeclarationNoIn, InitialiserNoIn and AssignmentExpressionNoIn are evaluated instead of the contained VariableDeclarationList, VariableDeclaration, Initialiser and AssignmentExpression, respectively.

12.3 Empty Statement
Syntax
EmptyStatement : ;

Semantics
The production EmptyStatement : ; is evaluated as follows:
1. Return (normal, empty, empty).

12.4 Expression Statement
Syntax
ExpressionStatement : [lookahead { {, function} ] Expression ;

Note that an ExpressionStatement cannot start with an opening curly brace because that might make it ambiguous with a Block. Also, an ExpressionStatement cannot start with the function keyword because that might make it ambiguous with a FunctionDeclaration.

Semantics
The production ExpressionStatement : [lookahead { {, function} ] Expression ; is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Return (normal, Result(2), empty).

12.5 The if Statement
Syntax
IfStatement :
    if ( Expression ) Statement else Statement
    if ( Expression ) Statement

Each else for which the choice of associated if is ambiguous shall be associated with the nearest possible if that would otherwise have no corresponding else.

Semantics
The production IfStatement : if ( Expression ) Statement else Statement is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, go to step 7.
5. Evaluate the first Statement.
6. Return Result(5).
7. Evaluate the second Statement.
8. Return Result(7).

The production IfStatement : if ( Expression ) Statement is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Call ToBoolean(Result(2)).
4. If Result(3) is false, return (normal, empty, empty).
5. Evaluate Statement.
6. Return Result(5).

12.5.1 Strict Mode Restrictions
In strict mode code a Statement that is part of an IfStatement production may not be a VariableStatement nor may it be a LabelledStatement whose Statement production is a VariableStatement. The LabelledStatement restriction also applies if such a VariableStatement is preceded by multiple labels.

12.6 Iteration Statements
An iteration statement consists of a header (which consists of a keyword and a parenthesised control construct) and a body (which consists of a Statement).

Syntax
IterationStatement :

do Statement while ( Expression ) ;
while ( Expression ) Statement
for ( var VariableDeclarationListNoIn; Expression ; Expression ) Statement
for ( LeftHandSideExpression in Expression ) Statement
for ( var VariableDeclarationNoIn in Expression ) Statement

Strict Mode Restrictions
A Statement that is an element of an IterationStatement production may not be a VariableStatement nor may it be a LabelledStatement whose Statement production is a VariableStatement. The LabelledStatement restriction also applies if such a VariableStatement is preceded by multiple labels.

12.6.1 The do-while Statement
The production do Statement while ( Expression ) ; is evaluated as follows:
1. Let V = empty.
2. Evaluate Statement.
3. If Result(2).value is not empty, let V = Result(2).value.
4. If Result(2).type is continue and Result(2).target is in the current label set, go to step 7.
5. If Result(2).type is break and Result(2).target is in the current label set, return (normal, V, empty).
6. If Result(2) is an abrupt completion, return Result(2).
7. Evaluate Expression.
8. Call GetValue(Result(7)).
9. Call ToBoolean(Result(8)).
10. If Result(9) is true, go to step 2.
11. Return (normal, V, empty);

12.6.2 The while statement
The production IterationStatement : while ( Expression ) Statement is evaluated as follows:
1. Let V = empty.
2. Evaluate Expression.
3. Call GetValue(Result(2)).
4. Call ToBoolean(Result(3)).
5. If Result(4) is false, return (normal, V, empty).
7. If Result(6).value is not empty, let V = Result(6).value.
8. If Result(6).type is continue and Result(6).target is in the current label set, go to 2.
9. If Result(6).type is break and Result(6).target is in the current label set, return (normal, V, empty).
10. If Result(6) is an abrupt completion, return Result(6).
11. Go to step 2.

12.6.3 The for Statement

The production IterationStatement : for (ExpressionNoInopt ; Expressionopt ; Expressionopt) Statement is evaluated as follows:

1. If ExpressionNoIn is not present, go to step 4.
2. Evaluate ExpressionNoIn.
3. Call GetValue(Result(2)). (This value is not used.)
4. Let V = empty.
5. If the first Expression is not present, go to step 10.
6. Evaluate the first Expression.
7. Call GetValue(Result(6)).
8. Call ToBoolean(Result(7)).
9. If Result(8) is false, go to step 19.
10. Evaluate Statement.
11. If Result(10).value is not empty, let V = Result(10).value
12. If Result(10).type is break and Result(10).target is in the current label set, go to step 19.
13. If Result(10).type is continue and Result(10).target is in the current label set, go to step 15.
14. If Result(10) is an abrupt completion, return Result(10).
15. If the second Expression is not present, go to step 5.
16. Evaluate the second Expression.
17. Call GetValue(Result(16)). (This value is not used.)
18. Go to step 5.

The production IterationStatement : for { var VariableDeclarationListNoIn ; Expressionopt ; Expressionopt } Statement is evaluated as follows:

1. Evaluate VariableDeclarationListNoIn.
2. Let V = empty.
3. If the first Expression is not present, go to step 8.
4. Evaluate the first Expression.
5. Call GetValue(Result(4)).
6. Call ToBoolean(Result(5)).
7. If Result(6) is false, go to step 17.
8. Evaluate Statement.
9. If Result(8).value is not empty, let V = Result(8).value.
10. If Result(8).type is break and Result(8).target is in the current label set, go to step 17.
11. If Result(8).type is continue and Result(8).target is in the current label set, go to step 13.
12. If Result(8) is an abrupt completion, return Result(8).
13. If the second Expression is not present, go to step 3.
14. Evaluate the second Expression.
15. Call GetValue(Result(14)). (This value is not used.)
16. Go to step 3.
17. Return (normal, V, empty).

12.6.4 The for-in Statement

The production IterationStatement : for { LeftHandSideExpression in Expression } Statement is evaluated as follows:
1. Evaluate the Expression.
2. Call GetValue(Result(1)).
3. If Result(2) is null or undefined, return (normal, V, empty).
4. Call ToObject(Result(2)).
5. Let V = empty.
6. Get the name of the next property of Result(4) whose [[Enumerable]] attribute is true. If there is no such property, go to step 15.
7. Evaluate the LeftHandSideExpression (it may be evaluated repeatedly).
8. Call PutValue(Result(6), Result(7)).
10. If Result(9).value is not empty, let V = Result(9).value.
11. If Result(2).type is break and Result(9).target is in the current label set, go to step 15.
12. If Result(2).type is continue and Result(9).target is in the current label set, go to step 16.
13. If Result(7) is an abrupt completion, return Result(9).
15. Return (normal, V, empty).

The production IterationStatement : for ( var VariableDeclarationNoIn in Expression ) Statement is evaluated as follows:
1. Evaluate VariableDeclarationNoIn.
2. Evaluate Expression.
3. CallGetValue(Result(2)).
4. If Result(3) is null or undefined, return (normal, V, empty).
5. Call ToObject(Result(3)).
7. Get the name of the next property of Result(5) whose [[Enumerable]] attribute is true. If there is no such property, go to step 16.
8. Evaluate Result(1) as if it were an Identifier; see [step 7 from the previous algorithm](it may be evaluated repeatedly).
9. Call PutValue(Result(7), Result(8)).
10. Evaluate Statement.
11. If Result(10).value is not empty, let V = Result(10).value.
12. If Result(2).type is break and Result(10).target is in the current label set, go to step 15.
13. If Result(2).type is continue and Result(10).target is in the current label set, go to step 16.
14. If Result(9) is an abrupt completion, return Result(9).
15. Go to step 7.

The mechanics of enumerating the properties (step 5 in the first algorithm, step 6 in the second) is not specified. Properties of the object being enumerated may be deleted during enumeration. If a property that has not yet been visited during enumeration is deleted, then it will not be visited. If new properties are added to the object being enumerated during enumeration, the newly added properties are guaranteed not to be visited in the active enumeration.

Enumerating the properties of an object includes enumerating properties of its prototype, and the prototype of the prototype, and so on, recursively; but a property of a prototype is not enumerated if it is “shadowed” because some previous object in the prototype chain has a property with the same name.

### 12.6.4.1 Strict Mode Restrictions

The same restrictions apply as specified in section 11.13.1.1.

### 12.7 The continue Statement

**Syntax**

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

**Semantics**

15 December 2008
A program is considered syntactically incorrect if either of the following are true:
The program contains a `continue` statement without the optional `Identifier`, which is not nested, directly or indirectly (but not crossing function boundaries), within a `IterationStatement`.
The program contains a `continue` statement with the optional `Identifier`, where `Identifier` does not appear in the label set of an enclosing (but not crossing function boundaries) `IterationStatement`.

A `ContinueStatement` without an `Identifier` is evaluated as follows:
1. Return `(continue, empty, empty)`.

A `ContinueStatement` with the optional `Identifier` is evaluated as follows:
1. Return `(continue, empty, Identifier)`.

### 12.8 The `break` Statement

**Syntax**

`BreakStatement : break [no LineTerminator here] Identifieropt ;`

**Semantics**

A program is considered syntactically incorrect if either of the following are true:
The program contains a `break` statement without the optional `Identifier`, which is not nested, directly or indirectly (but not crossing function boundaries), within an `IterationStatement` or a `SwitchStatement`.
The program contains a `break` statement with the optional `Identifier`, where `Identifier` does not appear in the label set of an enclosing (but not crossing function boundaries) `Statement`.

A `BreakStatement` without an `Identifier` is evaluated as follows:
1. Return `(break, empty, empty)`.

A `BreakStatement` with an `Identifier` is evaluated as follows:
1. Return `(break, empty, Identifier)`.

### 12.9 The `return` Statement

**Syntax**


**Semantics**

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

The production `ReturnStatement : return [no LineTerminator here] Expressionopt ;` is evaluated as:

1. If the `Expression` is not present, return `(return, undefined, empty)`.
2. Evaluate `Expression`.
3. Call `GetValue(Result(2))`.
4. Return `(return, Result(3), empty)`.

### 12.10 The `with` Statement

**Syntax**

`WithStatement : with ( Expression ) Statement`
Description
The with statement adds a computed object environment record to the lexical environment of the current execution context, then executes a statement with this augmented scope chain, then restores the lexical environment.

Semantics
The production WithStatement : with ( Expression ) Statement is evaluated as follows:
1. Let val be the result of evaluating Expression.
2. Let obj be ToObject(GetValue(val)).
3. Let oldEnv be the running execution context’s LexicalEnvironment.
4. Let newEnv be the result of calling NewObjectEnvironmentRecord(O,E) passing obj and oldEnv as the arguments.
5. Set the running execution context’s LexicalEnvironment to newEnv.
6. Let C be the result of evaluating Statement but if an exception is thrown during the evaluation, let C be (throw, V, empty), where V is the exception. (Execution now proceeds as if no exception were thrown.)
7. Set the running execution context’s Lexical Environment to oldEnv.
8. Return C.

NOTE
No matter how control leaves the embedded ‘Statement’, whether normally or by some form of abrupt completion or exception, the LexicalEnvironment is always restored to its former state.

12.10.1 Strict Mode Restrictions
Strict mode code may not include a WithStatement. The occurrence of a WithStatement in such a context is treated as a syntax error.

12.11 The switch Statement
Syntax

SwitchStatement : switch ( Expression ) CaseBlock

CaseBlock : { CaseClausesopt }

CaseClauses : CaseClause CaseClauses CaseClause

CaseClause : case Expression : StatementListopt

DefaultClause : default : StatementListopt

Semantics
The production SwitchStatement : switch ( Expression ) CaseBlock is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Evaluate CaseBlock, passing it Result(2) as a parameter.
4. If Result(3).type is break and Result(3).target is in the current label set, return (normal, Result(3).value, empty).
5. Return Result(3).
The production \textit{CaseBlock} : \{ CaseClauses_{opt} \} is given an input parameter, \textit{input}, and is evaluated as follows:

1. Let \( V = \text{empty} \).
2. Let \( A \) be the list of CaseClause items in source text order.
3. Let \( C \) be the next CaseClause in \( A \). If there is no such CaseClause, then go to step 16.
4. Evaluate \( C \).
5. If \textit{input} is not equal to Result(4) as defined by the \(!=\) operator, then go to step 3.
6. If \( C \) does not have a StatementList, then go to step 10.
7. Evaluate \( C \)'s StatementList and let \( R \) be the result.
8. If \( R \) is an abrupt completion, then return \( R \).
9. Let \( V = R \).value.
10. Let \( C \) be the next CaseClause in \( A \). If there is no such CaseClause, then go to step 16.
11. If \( C \) does not have a StatementList, then go to step 10.
12. Evaluate \( C \)'s StatementList and let \( R \) be the result.
13. If \( R \).value is not \text{empty}, then let \( V = R \).value.
14. If \( R \) is an abrupt completion, then return \((R.type, V, R.target)\).
15. Go to step 10.
16. Return \((\text{normal}, V, \text{empty})\).

The production \textit{CaseBlock} : \{ CaseClauses_{opt}, DefaultClause CaseClauses_{opt} \} is given an input parameter, \textit{input}, and is evaluated as follows:

1. Let \( V = \text{empty} \).
2. Let \( A \) be the list of CaseClause items in the first CaseClauses, in source text order.
3. Let \( C \) be the next CaseClause in \( A \). If there is no such CaseClause, then go to step 11.
4. Evaluate \( C \).
5. If \textit{input} is not equal to Result(4) as defined by the \(!=\) operator, then go to step 3.
6. If \( C \) does not have a StatementList, then go to step 20.
7. Evaluate \( C \)'s StatementList and let \( R \) be the result.
8. If \( R \) is an abrupt completion, then return \( R \).
9. Let \( V = R \).value.
11. Let \( B \) be the list of CaseClause items in the second CaseClauses, in source text order.
12. Let \( C \) be the next CaseClause in \( B \). If there is no such CaseClause, then go to step 26.
13. Evaluate \( C \).
14. If \textit{input} is not equal to Result(13) as defined by the \(!=\) operator, then go to step 12.
15. If \( C \) does not have a StatementList, then go to step 31.
16. Evaluate \( C \)'s StatementList and let \( R \) be the result.
17. If \( R \) is an abrupt completion, then return \( R \).
18. Let \( V = R \).value.
20. Let \( C \) be the next CaseClause in \( A \). If there is no such CaseClause, then go to step 26.
21. If \( C \) does not have a StatementList, then go to step 20.
22. Evaluate \( C \)'s StatementList and let \( R \) be the result.
23. If \( R \).value is not \text{empty}, then let \( V = R \).value.
24. If \( R \) is an abrupt completion, then return \((R.type, V, R.target)\).
26. If the DefaultClause does not have a StatementList, then go to step 30.
27. Evaluate the DefaultClause's StatementList and let \( R \) be the result.
28. If \( R \).value is not \text{empty}, then let \( V = R \).value.
29. If \( R \) is an abrupt completion, then return \((R.type, V, R.target)\).
30. Let \( B \) be the list of CaseClause items in the second CaseClauses, in source text order.
31. Let \( C \) be the next CaseClause in \( B \). If there is no such CaseClause, then go to step 37.
32. If \( C \) does not have a StatementList, then go to step 31.
33. Evaluate \( C \)'s StatementList and let \( R \) be the result.
34. If \( R \).value is not \text{empty}, then let \( V = R \).value.
35. If \( R \) is an abrupt completion, then return \((R.type, V, R.target)\).
36. Go to step 31.
37. Return \((\text{normal}, V, \text{empty})\).
The production **CaseClause** : `case` Expression : StatementList, is evaluated as follows:

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

**NOTE**

Evaluating CaseClause does not execute the associated StatementList. It simply evaluates the Expression and returns the value, which the CaseBlock algorithm uses to determine which StatementList to start executing.

### 12.11.1 Strict Mode Restrictions

A Statement that is an element of an StatementList that is part of a CaseClause or DefaultClause may not be a `VariableStatement` nor may it be a `LabelledStatement` whose Statement production is a `VariableStatement`. The `LabelledStatement` restriction also applies if such a `VariableStatement` is preceded by multiple labels.

### 12.12 Labelled Statements

**Syntax**

`LabelledStatement` :

Identifier : Statement

**Semantics**

A Statement may be prefixed by a label. Labelled statements are only used in conjunction with labelled `break` and `continue` statements. ECMAScript has no `goto` statement.

An ECMAScript program is considered syntactically incorrect if it contains a `LabelledStatement` that is enclosed by a `LabelledStatement` with the same Identifier as label. This does not apply to labels appearing within the body of a `FunctionDeclaration` that is nested, directly or indirectly, within a labelled statement. The production `Identifier : Statement` is evaluated by adding `Identifier` to the label set of `Statement` and then evaluating `Statement`. If the `LabelledStatement` itself has a non-empty label set, these labels are also added to the label set of `Statement` before evaluating it. If the result of evaluating `Statement` is `(break, V, L)` where `L` is equal to `Identifier`, the production results in `(normal, V, empty)`.

Prior to the evaluation of a `LabelledStatement`, the contained `Statement` is regarded as possessing an empty label set, except if it is an `IterationStatement` or a `SwitchStatement`, in which case it is regarded as possessing a label set consisting of the single element, `empty`.

### 12.13 The `throw` statement

**Syntax**

`ThrowStatement` :

`throw` [no LineTerminator here] Expression ;

**Semantics**

The production `ThrowStatement : throw` [no LineTerminator here] Expression ; is evaluated as:

1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Return (throw, Result(2), empty).

### 12.14 The `try` statement

**Syntax**

`TryStatement` :

`try` Block Catch
`try` Block Finally
`try` Block Catch Finally
Catch:
  catch (Identifier) Block

Finally:
  finally Block

Description
The `try` statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a `throw` statement. The `catch` clause provides the exception-handling code. When a catch clause catches an exception, its `Identifier` is bound to that exception.

Semantics
The production `TryStatement : try Block Catch` is evaluated as follows:
1. Evaluate `Block`.
2. If `Result(1).type` is not `throw`, return `Result(1)`.
3. Evaluate `Catch` with parameter `Result(1)`.
4. Return `Result(3)`.

The production `TryStatement : try Block Finally` is evaluated as follows:
1. Evaluate `Block`.
2. Evaluate `Finally`.
3. If `Result(2).type` is `normal`, return `Result(1)`.
4. Return `Result(2)`.

The production `TryStatement : try Block Catch Finally` is evaluated as follows:
1. Evaluate `Block`.
2. Let `C = Result(1)`.
3. If `Result(1).type` is not `throw`, go to step 6.
4. Evaluate `Catch` with parameter `Result(1)`.
5. Let `C = Result(4)`.
6. Evaluate `Finally`.
7. If `Result(6).type` is `normal`, return `C`.
8. Return `Result(6)`.

The production `Catch : catch (Identifier) Block` is evaluated as follows:
1. Let `C` be the parameter that has been passed to this production.
2. Let `oldEnv` be the running execution context’s `LexicalEnvironment`
3. Let `catchEnv` be the result of calling `NewDeclarativeEnvironmentRecord(E)` passing `oldEnv` as the argument.
4. Call the `CreateMutableBinding(N)` concrete method of `catchEnv` passing the `Identifier` String value as the arguments. Note that the last argument is immaterial in this situation.
5. Set the running execution context’s `LexicalEnvironment` to `catchEnv`.
6. Let `B` be the result of evaluating `Block`.
7. Return `B`.

The production `Finally : finally Block` is evaluated as follows:
1. Evaluate `Block`.
2. Return `Result(1)`.

12.15 Debugger statement

Syntax

15 December 2008
DebuggerStatement:
  debugger;

Semantics
Evaluating the DebuggerStatement production may allow an implementation to cause a breakpoint when run under a debugger.

13 Function Definition

Syntax

FunctionDeclaration:
  function Identifier ( FormalParameterList?, ) { FunctionBody }

FunctionExpression:
  function Identifier?, ( FormalParameterList?, ) { FunctionBody }

FormalParameterList:
  Identifier FormalParameterList?, Identifier

FunctionBody:
  UseStrictDirective?, SourceElements

Semantics
The production FunctionDeclaration: function Identifier ( FormalParameterList?, ) { FunctionBody } is processed for function declarations as follows during Declaration Binding instantiation (10.3.3):

1. Return the result of creating a new Function object as specified in 13.2 with parameters specified by FormalParameterList?, and body specified by FunctionBody. Pass in the VariableEnvironment of the running execution context as the Scope and the string value of Identifier as Name.

The production FunctionExpression: function ( FormalParameterList?, ) { FunctionBody } is evaluated as follows:

1. Return the result of creating a new Function object as specified in 13.2 with parameters specified by FormalParameterList?, and body specified by FunctionBody. Pass in the LexicalEnvironment of the running execution context as the Scope and an empty string as Name.

The production FunctionExpression: function Identifier ( FormalParameterList?, ) { FunctionBody } is evaluated as follows:

1. Let funcEnv be the result of calling NewDeclarativEnvironmentRecord(E) passing the running execution context’s Lexical Environment as the argument
2. Let envRec be funcEnv’s environment record.
3. Call the CreateImmutableBinding(N) concrete method of envRec passing the string value of Identifier as the argument.
4. Let closure be the result of creating a new Function object as specified in 13.2 with parameters specified by FormalParameterList?, and body specified by FunctionBody. Pass in funcEnv as the Scope and the string value of Identifier as Name.
5. Call the InitializeImmutableBinding(N,V) concrete method of envRec passing the string value of Identifier and closure as the arguments.

NOTE
The Identifier in a FunctionExpression can be referenced from inside the FunctionExpression’s FunctionBody to allow the function to call itself recursively. However, unlike in a FunctionDeclaration, the Identifier in a FunctionExpression cannot be referenced from and does not affect the scope enclosing the FunctionExpression.

The production FunctionBody: SourceElements is evaluated as follows:
13.1 Definitions

This section is no longer used.

13.2 Creating Function Objects

Given an optional parameter list specified by `FormalParameterList`, a body specified by `FunctionBody`, a `Lexical Environment` specified by `Scope`, and a possibly empty string `Name`, a Function object is constructed as follows:

1. Create a new native ECMAScript object and let `F` be that object.
2. Set the `[[Class]]` internal property of `F` to "Function".
3. Set the `[[Prototype]]` internal property of `F` to the original Function prototype object as specified in 15.3.3.1.
4. Set the `[[Call]]` internal property of `F` as described in 13.2.1.
5. Set the `[[Construct]]` internal property of `F` as described in 13.2.2.
6. Set the `[[Scope]]` internal property of `F` to the value of `Scope`.
7. Let `names` be a List containing, in left to right textual order, the strings corresponding to the identifiers of `FormalParameterList`.
8. Set the `[[FormalParameters]]` internal property of `F` to `names`.
9. Set the `[[Code]]` internal property of `F` to `FunctionBody`.
10. Set the `length` property of `F` to the number of formal parameters specified in `FormalParameterList`. If no parameters are specified, set the `length` property of `F` to 0. This property is given attributes as specified in 15.3.5.1.
11. Set the `[[Extensible]]` internal property of `F` to true.
12. Let `proto` be the result of creating a new object as would be constructed by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
13. Set the `[[Prototype]]` property of `proto` to `F`. This property has attributes `{ [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true }`.
14. Set the `[[Prototype]]` property of `F` to `proto`. This property is given attributes as specified in 15.3.5.2.
15. Set the `[[name]]` property of `F` to `Name`. This property is given attributes as specified in 15.3.5.4.

NOTE

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

13.2.1 `[[Call]]`

When the `[[Call]]` property for a Function object `F` is called, the following steps are taken:

1. Let `funcCtx` be the result of establishing a new execution context for function code using `F`'s `FormalParameterList`, the passed arguments list `args`, and the `this` value as described in 10.4.3.
2. Let `result` be the result of evaluating `F`'s `FunctionBody`.
3. Exit the execution context `funcCtx`, restoring the previous execution context.
4. If `result.type` is `throw` then throw `result.value`.
5. If `result.type` is `return` then return `result.value`.
6. Otherwise `result.type` must be `normal`, return `undefined`.

13.2.2 `[[Construct]]`

When the `[[Construct]]` property for a Function object `F` is called, the following steps are taken:

1. Let `obj` be a newly created, native ECMAScript object.
2. Set the `[[Class]]` internal property of `obj` to "Object".
3. Set the `[[Extensible]]` internal property of `obj` to true.

15 December 2008
4. Let `proto` be the value of the `prototype` property of `F`.
5. If `proto` is an `Object`, set the `[[Prototype]]` internal property of `obj` to `proto`.
6. If `proto` is not an `Object`, set the `[[Prototype]]` internal property of `obj` to the standard built-in Object prototype object as described in 15.2.4.
7. Let `result` be the result of invoking the `[[Call]]` internal property of `F`, providing `obj` as the `this` value and providing the argument list passed into `[[Construct]]` as `args`.
8. If `Type(result)` is `Object` then return `result`.
14 Program Syntax

Program : UseStrictDirective opt SourceElements

UseStrictDirective : "use strict useExtension opt" ;

useExtension : , DoubleStringCharacters opt

SourceElements : SourceElement SourceElements SourceElement

SourceElement : Statement FunctionDeclaration

Semantics

The production Program : UseStrictDirective opt SourceElements is evaluated as follows:

1. If the optional UseStrictDirective is present, SourceElements is processed and evaluated in the following steps as strict mode code. Otherwise SourceElements is processed and evaluated in the following steps as non-strict mode code.
3. Evaluate SourceElements.
4. Return Result(3).

The production SourceElements : SourceElement is evaluated as follows:

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

The production SourceElements : SourceElements SourceElement is evaluated as follows:

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

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

The production SourceElement : Statement is evaluated as follows:

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

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

1. Process FunctionDeclaration for function declarations (see clause 13).

15 December 2008
The production `SourceElement : FunctionDeclaration` is evaluated as follows:
1. Return `(normal, empty, empty)`.

The productions `UseStrictDirective_opt : " use strict useExtension_opt" ;` and `useExtension_opt : , DoubleStringCharacters_opt` have no associated semantic actions.
15 Native ECMAScript Objects

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

Unless specified otherwise, the [[Class]] property of a built-in object is "Function" if that built-in object has a [[Call]] property, or "Object" if that built-in object does not have a [[Call]] property. Unless specified otherwise, the [[Extensible]] property of a built-in object has the value true.

Many built-in objects are functions: they can be invoked with arguments. Some of them furthermore are constructors: they are functions intended for use with the new operator. For each built-in function, this specification describes the arguments required by that function and properties of the Function object. For each built-in constructor, this specification furthermore describes properties of the prototype object of that constructor and properties of specific object instances returned by a new expression that invokes that constructor.

Unless otherwise specified in the description of a particular function, if a function or constructor described in this section is given fewer arguments than the function is specified to require, the function or constructor shall behave exactly as if it had been given sufficient additional arguments, each such argument being the undefined value.

Unless otherwise specified in the description of a particular function, if a function or constructor described in this section is given more arguments than the function is specified to allow, the behaviour of the function or constructor is undefined.

NOTE Implementations that add additional capabilities to the set of built-in functions are encouraged to do so by adding new functions rather than adding new parameters to existing functions.

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

Every built-in prototype object has the Object prototype object, which is the initial value of the expression Object.prototype (15.3.2.1), as the value of its internal [[Prototype]] property, except the Object prototype object itself.

None of the built-in functions described in this section shall implement the internal [[Construct]] method unless otherwise specified in the description of a particular function. None of the built-in functions described in this section shall initially have a prototype property unless otherwise specified in the description of a particular function. Every built-in Function object described in this section—whether as a constructor, an ordinary function, or both—has a length property whose value is an integer. Unless otherwise specified, this value is equal to the largest number of named arguments shown in the section headings for the function description, including optional parameters.

NOTE For example, the Function object that is the initial value of the slice property of the String prototype object is described under the section heading ―String.prototype.slice (start, end)‖ which shows the two named arguments start and end; therefore the value of the length property of that Function object is 2.

In every case, the length property of a built-in Function object described in this section has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false } (and no others). Every other property described in this section has the attribute { [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: true } unless otherwise specified.

15.1 The Global Object

The unique global object is created before control enters any execution context.

Unless otherwise specified, the properties of the global object have attributes { [[Enumerable]]: false }.

The global object does not have a [[Construct]] property; it is not possible to use the global object as a constructor with the new operator.
The global object does not have a `[[Call]]` property; it is not possible to invoke the global object as a function.

The values of the `[[Prototype]]` and `[[Class]]` properties of the global object are implementation-dependent.

In addition to the properties defined in this specification the global object may have additional host defined properties. This may include a property whose value is the global object itself; for example, in the HTML document object model the `window` property of the global object is the global object itself.

### 15.1 Value Properties of the Global Object

#### 15.1.1 NaN

The initial value of `NaN` is `NaN`. This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

#### 15.1.2 Infinity

The initial value of `Infinity` is `+0`. This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

#### 15.1.3 undefined

The initial value of `undefined` is `undefined`. This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

### 15.1.2 Function Properties of the Global Object

#### 15.1.2.1 eval (x)

When the `eval` function is called with one argument `x`, the following steps are taken:

1. If `x` is not a string value, return `x`.
2. Let `prog` be the ECMAScript code that is the result of parsing `x` as a `Program`. If the parse fails, throw a `SyntaxError` exception (but see also clause 16).
3. Let `evalCtx` be the result of establishing a new execution context (10.4.2) for the eval code `prog`.
4. Let `result` be the result of evaluating the program `prog`.
5. Exit the running execution context `evalCtx`, restoring the previous execution context.
6. If `result` is normal and its completion value is a value `V`, then return the value `V`.
7. If `result.type` is `normal` and its completion value is empty, then return the value `undefined`.
8. Otherwise, `result.type` must be throw. Throw `result.value` as an exception.

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

#### 15.1.2.1 Strict Mode Restrictions

If strict mode code uses the value of the `eval` property any way other than as a direct call (that is, other than by the explicit use of its name as an `Identifier` which is the `MemberExpression` in a `CallExpression`), or if the `eval` property is assigned to, an `EvalError` exception is thrown.

#### 15.1.2.2 parseInt (string, radix)

The `parseInt` function produces an integer value dictated by interpretation of the contents of the `string` argument according to the specified `radix`. Leading white space in the string is ignored. If `radix` is `undefined` or 0, it is assumed to be 10 except when the number begins with the character pairs `0x` or `0X`, in which case a radix of 16 is assumed. Any radix-16 number may also optionally begin with the character pairs `0x` or `0X`.

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

1. Call `ToStrings(string)`.
2. Let `S` be a newly created substring of `Result(1)` consisting of the first character that is not a `StrWhiteSpaceChar` and all characters following that character. (In other words, remove leading white space.)
3. Let `sign` be 1.
4. If `S` is not empty and the first character of `S` is a minus sign `−`, let `sign` be −1.

15 December 2008
5. If \( S \) is not empty and the first character of \( S \) is a plus sign \( + \) or a minus sign \( - \), then remove the first character from \( S \).
6. Let \( R = \text{ToInt32}(\text{radix}) \).
7. If \( R = 0 \), go to step 11.
8. If \( R < 2 \) or \( R > 36 \), then return \( \text{NaN} \).
9. If \( R = 16 \), go to step 13.
11. Let \( R = 10 \).
12. If the length of \( S \) is at least 1 and the first character of \( S \) is \( \text{"0"} \), then at the implementation’s discretion either let \( R = 8 \) or leave \( R \) unchanged.
13. If the length of \( S \) is at least 2 and the first two characters of \( S \) are either \( \text{"0x"} \) or \( \text{"0X"} \), then remove the first two characters from \( S \) and let \( R = 16 \).
14. If \( S \) contains any character that is not a radix-\( R \) digit, then let \( Z \) be the substring of \( S \) consisting of all characters before the first such character; otherwise, let \( Z \) be \( S \).
15. If \( Z \) is empty, return \( \text{NaN} \).
16. Compute the mathematical integer value that is represented by \( Z \) in radix-\( R \) notation, using the letters \( A-Z \) and \( a-z \) for digits with values 10 through 35. (However, if \( R \) is 10 and \( Z \) contains more than 20 significant digits, every significant digit after the 20th may be replaced by a \( 0 \) digit, at the option of the implementation; and if \( R \) is not 2, 4, 8, 10, 16, or 32, then Result(16) may be an implementation-dependent approximation to the mathematical integer value that is represented by \( Z \) in radix-\( R \) notation.)
17. Compute the number value for Result(16).
18. Return \( \text{sign} \) \( \times \) Result(17).

**NOTE**
\( \text{parseInt} \) may interpret only a leading portion of the string as an integer value; it ignores any characters that cannot be interpreted as part of the notation of an integer, and no indication is given that any such characters were ignored.

When \( \text{radix} \) is 0 or \( \text{undefined} \) and the string’s number begins with a \( 0 \) digit not followed by an \( x \) or \( X \), then the implementation may, at its discretion, interpret the number either as being octal or as being decimal. Implementations are encouraged to interpret numbers in this case as being decimal.

15.1.2.3
\( \text{parseFloat} \) (string)
The \( \text{parseFloat} \) function produces a number value dictated by interpretation of the contents of the string argument as a decimal literal.

When the \( \text{parseFloat} \) function is called, the following steps are taken:
1. Call \( \text{ToString}(\text{string}) \).
2. Compute a substring of Result(1) consisting of the leftmost character that is not a \( \text{StrWhiteSpaceChar} \) and all characters to the right of that character. (In other words, remove leading white space.)
3. If neither Result(2) nor any prefix of Result(2) satisfies the syntax of a \( \text{StrDecimalLiteral} \) (see 9.3.1), return \( \text{NaN} \).
4. Compute the longest prefix of Result(2), which might be Result(2) itself, which satisfies the syntax of a \( \text{StrDecimalLiteral} \).
5. Return the number value for the MV of Result(4).

**NOTE**
\( \text{parseFloat} \) may interpret only a leading portion of the string as a number value; it ignores any characters that cannot be interpreted as part of the notation of a decimal literal, and no indication is given that any such characters were ignored.

15.1.2.4
is\( \text{NaN} \) (number)
Returns \( \text{true} \) if the result is \( \text{NaN} \), and otherwise returns \( \text{false} \).
1. Call \( \text{GetValue}(\text{number}) \).
2. Call \( \text{ToNumber}(\text{Result}(1)) \).
3. If Result(2) is \( \text{NaN} \), return \( \text{true} \).
4. Return \( \text{false} \).
15.1.2.5 isFinite (number)

Returns false if the result is NaN, +\infty, or -\infty, and otherwise returns true.

1. Call GetValue(number).
2. Call ToNumber(Result(1)).
3. If Result(2) is NaN, +\infty, or -\infty, return false.
4. Return true.

15.1.3 URI Handling Function Properties

Uniform Resource Identifiers, or URIs, are strings that identify resources (e.g. web pages or files) and transport protocols by which to access them (e.g. HTTP or FTP) on the Internet. The ECMAScript language itself does not provide any support for using URIs except for functions that encode and decode URIs as described in 15.1.3.1, 15.1.3.2, 15.1.3.3 and 15.1.3.4.

NOTE
Many implementations of ECMAScript provide additional functions and methods that manipulate web pages; these functions are beyond the scope of this standard.

A URI is composed of a sequence of components separated by component separators. The general form is:

```
Scheme : First / Second ; Third ? Fourth
```

where the italicised names represent components and the "-", ":", "/", ";", and "?" are reserved characters used as separators. The encodeURI and decodeURI functions are intended to work with complete URIs; they assume that any reserved characters in the URI are intended to have special meaning and so are not encoded. The encodeURIComponent and decodeURIComponent functions are intended to work with the individual component parts of a URI; they assume that any reserved characters represent text and so must be encoded so that they are not interpreted as reserved characters when the component is part of a complete URI.

The following lexical grammar specifies the form of encoded URIs.

```
uri ::= uriCharactersopt
uriCharacters ::= uriCharacter uriCharactersopt
uriCharacter ::= uriReserved
uriUnescaped
uriEscaped
uriReserved ::= one of "; / ? : @ & = + $ ,
uriUnescaped ::= uriAlpha
DecimalDigit
uriMark
uriEscaped ::= % HexDigit HexDigit
uriAlpha ::= one of a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
uriMark ::= one of - _ . ! * ( )
```

15 December 2008
When a character to be included in a URI is not listed above or is not intended to have the special meaning sometimes given to the reserved characters, that character must be encoded. The character is first transformed into a sequence of octets using the UTF-8 transformation, with surrogate pairs first transformed from their UCS-2 to UCS-4 encodings. (Note that for code points in the range [0,127] this results in a single octet with the same value.) The resulting sequence of octets is then transformed into a string with each octet represented by an escape sequence of the form “%xx”.

The encoding and escaping process is described by the hidden function Encode taking two string arguments string and unescapedSet. This function is defined for expository purpose only.

1. Compute the number of characters in string.
2. Let R be the empty string.
3. Let k be 0.
4. If k equals Result(1), return R.
5. Let C be the character at position k within string.
6. If C is not in unescapedSet, go to step 9.
7. Let S be a string containing only the character C.
9. If the code point value of C is not less than 0xDC00 and not greater than 0xDFFF, throw a URIError exception.
10. If the code point value of C is less than 0xD800 or greater than 0xDBFF, let V be the code point value of C and go to step 16.
11. Increase k by 1.
12. If k equals Result(1), throw a URIError exception.
13. Get the code point value of the character at position k within string.
14. If Result(13) is less than 0xDC00 or greater than 0xDFFF, throw a URIError exception.
15. Let V be (((the code point value of C) – 0xD800) * 0x400 + (Result(13) – 0xDC00) + 0x10000).
16. Let Octets be the array of octets resulting by applying the UTF-8 transformation to V, and let L be the array size.
17. Let j be 0.
18. Get the value at position j within Octets.
19. Let S be a string containing three characters “%XY” where XY are two uppercase hexadecimal digits encoding the value of Result(18).
20. Let R be a new string value computed by concatenating the previous value of R and S.
22. If j is equal to L, go to step 25.
23. Go to step 18.
24. Let R be a new string value computed by concatenating the previous value of R and S.
25. Increase k by 1.

The unescaping and decoding process is described by the hidden function Decode taking two string arguments string and unescapedSet. This function is defined for expository purpose only.

1. Compute the number of characters in string.
2. Let R be the empty string.
3. Let k be 0.
4. If k equals Result(1), return R.
5. Let C be the character at position k within string.
6. If C is not ‘%’, go to step 40.
7. Let start be k.
8. If k + 2 is greater than or equal to Result(1), throw a URIError exception.
9. If the characters at position (k+1) and (k + 2) within string do not represent hexadecimal digits, throw a URIError exception.
10. Let B be the 8-bit value represented by the two hexadecimal digits at position (k + 1) and (k + 2).
11. Increment k by 2.
12. If the most significant bit in B is 0, let C be the character with code point value B and go to step 37.
13. Let n be the smallest non-negative number such that (B << n) & 0x80 is equal to 0.
14. If n equals 1 or n is greater than 4, throw a URIError exception.
15. Let Octets be an array of 8-bit integers of size n.
16. Put $B$ into Octets at position 0.
17. If $k + (3 \times (n - 1))$ is greater than or equal to Result(1), throw a URIError exception.
18. Let $j$ be 1.
19. $j$ equals $n$, go to step 29.
20. Increment $k$ by 1.
21. If the character at position $k$ is not “%”, throw a URIError exception.
22. Let $j$ be 1.
23. If $j$ equals $n$, go to step 29.
24. Increment $k$ by 1.
25. If the character at position $k$ is not ‟%”, throw a URIError exception.
26. If the characters at position $(k + 1)$ and $(k + 2)$ within string do not represent hexadecimal digits, throw a URIError exception.
27. Let $B$ be the 8-bit value represented by the two hexadecimal digits at position $(k + 1)$ and $(k + 2)$.
28. Increment $k$ by 2.
29. Put $B$ into Octets at position $j$.
30. Increment $j$ by 1.
32. Let $V$ be the value obtained by applying the UTF-8 transformation to Octets, that is, from an array of octets into a 32-bit value.
33. If $V$ is less than 0x10000, go to step 36.
34. If $V$ is greater than 0x10FFFF, throw a URIError exception.
35. Let $L$ be $(((V - 0x10000) & 0x3FF) + 0xDC00)$.
36. Let $H$ be $(((V - 0x10000) >> 10) & 0x3FF) + 0xD800)$.
37. Let $S$ be the string containing the two characters with code point values $H$ and $L$.
38. Go to step 41.
39. Let $C$ be the character with code point value $V$.
40. $C$ is not in reservedSet, go to step 40.
41. Let $S$ be the string containing only the character $C$.
42. Increase $k$ by 1.
43. Go to step 4.

NOTE 1
The syntax of Uniform Resource Identifiers is given in RFC2396.

NOTE 2
A formal description and implementation of UTF-8 is given in the Unicode Standard, Version 2.0, Appendix A.

In UTF-8, characters are encoded using sequences of 1 to 6 octets. The only octet of a "sequence" of one has the higher-order bit set to 0, the remaining 7 bits being used to encode the character value. In a sequence of $n$ octets, $n \geq 1$, the initial octet has the $n$ higher-order bits set to 1, followed by a bit set to 0. The remaining bits of that octet contain bits from the value of the character to be encoded. The following octets all have the higher-order bit set to 1 and the following bit set to 0, leaving 6 bits in each to contain bits from the character to be encoded. The possible UTF-8 encodings of ECMAScript characters are:

<table>
<thead>
<tr>
<th>Code Point Value</th>
<th>Representation</th>
<th>1st Octet</th>
<th>2nd Octet</th>
<th>3rd Octet</th>
<th>4th Octet</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000 - 0x007F</td>
<td>00000000 0zzzzzz</td>
<td>0zzzzzz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800 - 0x07FF</td>
<td>0000yyyy yyyyyyy</td>
<td>110yyyyy</td>
<td>10zzzzzz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0800 - 0x07FF</td>
<td>xxxxyyyy yyyyzzzz</td>
<td>1110xxxx</td>
<td>10yyyyyy</td>
<td>10zzzzzz</td>
<td></td>
</tr>
<tr>
<td>0x0800 - 0x07FF</td>
<td>11011vv vvvvxxx</td>
<td>11110uuu</td>
<td>10uuuuuu</td>
<td>10xxyyyy</td>
<td>10zzzzzz</td>
</tr>
<tr>
<td>0x0800 - 0x07FF</td>
<td>followed by</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0000 - 0xC0FF</td>
<td>110111yy yyyyzzzz</td>
<td>11110uuu</td>
<td>10uuuuuu</td>
<td>10xxyyyy</td>
<td>10zzzzzz</td>
</tr>
<tr>
<td>0x0800 - 0x07FF</td>
<td>not followed by</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>causes URIError</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Where 

\[ uuuu = vvvv + 1 \]

to account for the addition of 0x10000 as in 3.7, Surrogates of the Unicode Standard version 2.0.

The range of code point values 0xD800-0xDBFF is used to encode surrogate pairs; the above transformation combines a UCS-2 surrogate pair into a UCS-4 representation and encodes the resulting 21-bit value in UTF-8. Decoding reconstructs the surrogate pair.

15.1.3.1 decodeURI (encodedURI)

The `decodeURI` function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the `encodeURI` function is replaced with the character that it represents. Escape sequences that could not have been introduced by `encodeURI` are not replaced.

When the `decodeURI` function is called with one argument `encodedURI`, the following steps are taken:

1. Call `ToString(encodedURI)`.
2. Let `reservedURISet` be a string containing one instance of each character valid in `uriReserved` plus “#”.
3. Call `Decode(Result(1), reservedURISet)`
4. Return Result(3).

**NOTE**

The character “#” is not decoded from escape sequences even though it is not a reserved URI character.

15.1.3.2 decodeURIComponent (encodedURIComponent)

The `decodeURIComponent` function computes a new version of a URI in which each escape sequence and UTF-8 encoding of the sort that might be introduced by the `encodeURIComponent` function is replaced with the character that it represents.

When the `decodeURIComponent` function is called with one argument `encodedURIComponent`, the following steps are taken:

1. Call `ToString(encodedURIComponent)`.
2. Let `reservedURIComponentSet` be the empty string.
3. Call `Decode(Result(1), reservedURIComponentSet)`
4. Return Result(3).

15.1.3.3 encodeURI (uri)

The `encodeURI` function computes a new version of a URI in which each instance of certain characters is replaced by one, two or three escape sequences representing the UTF-8 encoding of the character.

When the `encodeURI` function is called with one argument `uri`, the following steps are taken:

1. Call `ToString(uri)`.
2. Let `unescapeURISet` be a string containing one instance of each character valid in `uriReserved` and `uriUnescaped` plus “#”.
3. Call `Encode(Result(1), unescapedURISet)`
4. Return Result(3).

**NOTE**
The character ‘#’ is not encoded to an escape sequence even though it is not a reserved or unescaped URI character.

15.1.3.4 encodeURIComponent (uriComponent)

The encodeURIComponent function computes a new version of a URI in which each instance of certain characters is replaced by one, two or three escape sequences representing the UTF-8 encoding of the character.

When the encodeURIComponent function is called with one argument uriComponent, the following steps are taken:

1. Call ToString(uriComponent).
2. Let unescapedURIComponentSet be a string containing one instance of each character valid in uriUnescaped.
3. Call Encode(Result(1), unescapedURIComponentSet)
4. Return Result(3).

15.1.4 Constructor Properties of the Global Object

15.1.4.1 Object ( . . . )
See 15.2.1 and 15.2.2.

15.1.4.2 Function ( . . . )
See 15.3.1 and 15.3.2.

15.1.4.3 Array ( . . . )
See 15.4.1 and 15.4.2.

15.1.4.4 String ( . . . )
See 15.5.1 and 15.5.2.

15.1.4.5 Boolean ( . . . )
See 15.6.1 and 15.6.2.

15.1.4.6 Number ( . . . )
See 15.7.1 and 15.7.2.

15.1.4.7 Date ( . . . )
See 15.9.2.

15.1.4.8 RegExp ( . . . )
See 15.10.3 and 15.10.4.

15.1.4.9 Error ( . . . )
See 15.11.1 and 15.11.2.

15.1.4.10 EvalError ( . . . )
See 15.11.6.1.

15.1.4.11 RangeError ( . . . )
See 15.11.6.2.

15.1.4.12 ReferenceError ( . . . )
See 15.11.6.3.

15.1.4.13 SyntaxError ( . . . )
See 15.11.6.4.

15.1.4.14 TypeError ( . . . )
See 15.11.6.5.

15.1.4.15 URIError ( . . . )
See 15.11.6.6.
15.1.5 Other Properties of the Global Object

15.1.5.1 Math
See 15.8.

15.1.5.2 JSON
See 15.12.

15.2 Object Objects

15.2.1 The Object Constructor Called as a Function
When `Object` is called as a function rather than as a constructor, it performs a type conversion.

15.2.1.1 Object ( [ value ] )
When the `Object` function is called with no arguments or with one argument `value`, the following steps are taken:
1. If `value` is null, `undefined` or not supplied, create and return a new Object object exactly if the object constructor had been called with the same arguments (15.2.2.1).
2. Return `ToObject(value)`.

15.2.2 The Object Constructor
When `Object` is called as part of a `new` expression, it is a constructor that may create an object.

15.2.2.1 `new Object ( [ value ] )`
When the `Object` constructor is called with no arguments or with one argument `value`, the following steps are taken:
1. If `value` is not supplied, go to step 8.
2. If the type of `value` is not Object, go to step 5.
3. If the `value` is a native ECMAScript object, do not create a new object but simply return `value`.
4. If the `value` is a host object, then actions are taken and a result is returned in an implementation-dependent manner that may depend on the host object.
5. If the type of `value` is String, return `ToObject(value)`.
6. If the type of `value` is Boolean, return `ToObject(value)`.
7. If the type of `value` is Number, return `ToObject(value)`.
8. (The argument `value` was not supplied or its type was Null or Undefined.) Create a new native ECMAScript object.
   The `[[Prototype]]` property of the newly constructed object is set to the Object prototype object.
   The `[[Class]]` property of the newly constructed object is set to "Object".
   The `[[Extensible]]` property of the newly constructed object is set to `true`.
   The newly constructed object has no `[[PrimitiveValue]]` property.
9. Return the newly created native object.

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

Besides the internal properties and the `length` property (whose value is 1), the Object constructor has the following properties:

15.2.3.1 `Object.prototype`
The initial value of `Object.prototype` is the Object prototype object (15.2.4).

This property has the attributes `{| [Writable]: false, [Enumerable]: false, [Configurable]: false |}`.

15.2.3.2 `Object.getPrototypeOf ( O )`
When the `getPrototypeOf` function is called with argument `O`, the following steps are taken:
1. If `Type(O)` is not Object throw a `TypeError` exception.
2. Return the `[[Prototype]]` internal property of `O`. 

Comment [pL11]: Deviations doc item 3.12 suggests removing this phrase.
15.2.3.3 Object.getOwnPropertyDescriptor ( O, P )
When the `Object.getOwnPropertyDescriptor` function is called, the following steps are taken:
1. If `Type(O)` is not Object throw a `TypeError` exception.
2. If `P` is `undefined` or `null`, let `name` be the empty string, otherwise let `name` be `ToString(P)`.
3. Let `desc` be the result of calling the `[[GetOwnProperty]]` internal method of `O` with argument `name`.
4. Return the result of calling `FromPropertyDescriptor(desc)`.

15.2.3.4 Object.getOwnPropertyNames ( O )
When the `Object.getOwnPropertyNames` function is called, the following steps are taken:
1. If `Type(O)` is not Object throw a `TypeError` exception.
2. Let `array` be the result of creating a new object as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.
3. For each named own property `P` of `O`
   a. Let `name` be the string value that is the name of `P`.
   b. Append a property to `array` as if by calling the standard built-in method `Array.prototype.push` with `array` as the `this` value, and with argument `name`.
4. Return `array`.

NOTE
If `O` is a `String` instance, the set of own properties processed in step 1a does not include the implicit properties defined in 15.5.5.2 that correspond to character positions of the object’s `[[PrimitiveValue]]` string.

15.2.3.5 Object.create ( O [, Properties] )
The `Object.create` function creates a new object with a specified prototype. When the `create` function is called, the following steps are taken:
1. If `Type(O)` is not Object throw a `TypeError` exception.
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. Set the `[[Prototype]]` internal property of `obj` to `O`.
4. Add own properties to `obj` as if by calling the standard built-in function `Object.defineProperties` with arguments `obj` and `Properties`.
5. Return `obj`.

15.2.3.6 Object.defineProperty ( O, P, Attributes )
The `defineProperty` function is used to add an own properties and/or update the attributes of existing own property of an object. When the `defineProperty` function is called, the following steps are taken:
1. If `Type(O)` is not Object throw a `TypeError` exception.
2. Let `name` be `ToString(P)`.
3. Let `desc` be the result of calling `ToPropertyDescriptor with Attributes` as the argument.
4. Call the `[[DefineOwnProperty]]` internal method of `O` with arguments `name`, `desc`, and `true`.
5. Return `O`.

15.2.3.7 Object.defineProperties ( O, Properties )
The `defineProperties` function is used to add own properties and/or update the attributes of existing own properties of an object. When the `defineProperties` function is called, the following steps are taken atomically:
1. If `Type(O)` is not Object throw a `TypeError` exception.
2. Let `props` be `ToObject(Properties)`.
3. For each named own property name `P` of `props`
   a. Let `descObj` be the result of calling the `[[GetOwnProperty]]` internal method of `props with P` as the argument.
   b. Let `desc` be the result of calling `ToPropertyDescriptor with descObj` as the argument.
4. Call the `[[DefineOwnProperty]]` internal method of `O` with arguments `P`, `desc`, and `true`.
5. Return `O`.

15 December 2008
The above algorithm is specified as a set of sequential steps that include the possibility of a exception being thrown as various intermediate points. Rather than failing after a partial update of O, this function must be implemented such that it either atomically completes all property updates successfully or fails without making any update to the properties of object O.

15.2.3.8 Object.seal ( O )
When the seal function is called, the following steps are taken:
1. If Type(O) is not Object throw a TypeError exception.
2. For each named own property name P of O,
   a. Let desc be the result of calling the [[GetOwnProperty]] method of O with P.
   b. If desc.[[Configurable]] is true, set desc.[[Configurable]] to false.
   c. Call the [[DefineOwnProperty]] internal method of O with P, desc, and true as arguments.
3. Set the internal [[Extensible]] internal property of O to false.
4. Return O.

The above algorithm is specified as a set of sequential steps that include the possibility of a exception being thrown as various intermediate points. Rather than failing after a partial update of O, this function must be implemented such that it either atomically completes all property updates successfully or fails without making any update to the properties of object O.

15.2.3.9 Object.freeze ( O )
When the freeze function is called, the following steps are taken:
1. If Type(O) is not Object throw a TypeError exception.
2. For each named own property name P of O,
   a. Let desc be the result of calling the [[GetOwnProperty]] method of O with P.
   b. If IsDataDescriptor(desc) then
      i. If desc.[[Writable]] is true, set desc.[[Writable]] to false.
      c. If desc.[[Configurable]] is true, set desc.[[Configurable]] to false.
      d. Call the [[DefineOwnProperty]] internal method of O with P, desc, and true as arguments.
3. Set the [[Extensible]] internal property of O to false.
4. Return O.

The above algorithm is specified as a set of sequential steps that include the possibility of a exception being thrown as various intermediate points. Rather than failing after a partial update of O, this function must be implemented such that it either atomically completes all updates successfully or fails without making any update to the properties of object O.

15.2.3.10 Object.preventExtensions ( O )
When the preventExtensions function is called, the following steps are taken:
    If Type(O) is not Object throw a TypeError exception.
Set the [[Extensible]] internal property of O to false.
Return O.

15.2.3.11 Object.isSealed ( O )
When the isSealed function is called with argument O, the following steps are taken:
1. If Type(O) is not Object throw a TypeError exception.
2. For each named own data property P of O,
   a. Let desc be the result of calling the [[GetOwnProperty]] internal method of O with P.
   b. If the desc.[[Configurable]] is true, then return false.
3. If the [[Extensible]] internal property of O is false, then return true.
4. Otherwise, return false.

15.2.3.12 Object.isFrozen ( O )
When the isFrozen function is called with argument O, the following steps are taken:
1. If Type(O) is not Object throw a TypeError exception.
2. For each named own data property name P of O,
   a. Let desc be the result of calling the [[GetOwnProperty]] method of O with P.
b. If IsDataDescriptor(desc) then
   i. If desc.[[Writable]] is true, return false.
 c. If desc.[[Configurable]] is true, then return false.
 3. If the [[Extensible]] internal property of O is true, then return false.
 4. Otherwise, return true.

15.2.13 Object.isExtensible (O)

When the isExtensible function is called with argument O, the following steps are taken:
1. If Type(O) is not Object throw a TypeError exception.
2. Return the Boolean value of the [[Extensible]] internal property of O.

15.2.14 Object.keys (O)

When the keys function is called with argument O, the following steps are taken:
1. If the Type(O) is not Object, throw a TypeError exception.
2. Let array be the result of creating a new Object as if by the expression new Array() where
   Array is the standard built-in constructor with that name.
3. For each own enumerable property of O, append the key string of the property to array.
4. Return array.

NOTE
If an implementation defines a specific order of enumeration for the for-in statement, Object.keys must
return that same order.

15.2.4 Properties of the Object Prototype Object

The value of the internal [[Prototype]] property of the Object prototype object is null, the value of the internal
[[Class]] property is "Object", and the value of the internal [[Extensible]] property is true.

15.2.4.1 Object.prototype.constructor

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

15.2.4.2 Object.prototype.toString ()

When the toString method is called, the following steps are taken:
1. Let O be the result of calling ToObject passing the this object as the argument.
2. Get the [[Class]] property of O.
3. Compute a string value by concatenating the three strings "{object ", Result(2), and "]".
4. Return Result(3).

NOTE 1
This function is provided to give all Objects a generic toString interface, even though not all may use it. Currently, Array, Number, and Date provide their own locale-sensitive toString methods.

NOTE 2
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.2.4.3 Object.prototype.toLocaleString ()

When the toLocaleString method is called, the following steps are taken:
1. Let O be the result of calling ToObject passing the this object as the argument.
2. Get the [[Class]] property of O.
3. If IsCallable(Result(2)) is false, throw a TypeError exception.
4. Call the [[Call]] internal method of Result(2) passing O as the this value and no arguments.
5. Return Result(4).

NOTE 1
This function returns the result of calling toString().

15.2.4.4 Object.prototype.valueOf ()

The valueOf method returns its this value. If the object is the result of calling the Object constructor
with a host object (15.2.2.1), it is implementation-defined whether valueOf returns its this value or
another value such as the host object originally passed to the constructor.
15.2.4.5 Object.prototype.hasOwnProperty (V)

When the `hasOwnProperty` method is called with argument `V`, the following steps are taken:

1. Let `O` be the result of calling `ToObject` passing the `this` object as the argument.
2. Call `ToString(V)`.
3. Call the `[[GetOwnProperty]]` internal method of `O` passing `Result(2)` as the argument.
4. If `Result(3)` is `undefined`, return `false`.
5. Return `true`.

**NOTE**
Unlike `[[HasProperty]]` (8.6.2.4), this method does not consider objects in the prototype chain.

15.2.4.6 Object.prototype.isPrototypeOf (V)

When the `isPrototypeOf` method is called with argument `V`, the following steps are taken:

1. Let `O` be the result of calling `ToObject` passing the `this` object as the argument.
2. If `V` is not an object, return `false`.
3. Let `V` be the value of the `[[Prototype]]` property of `V`.
4. If `V` is `null`, return `false`.
5. If `O` and `V` refer to the same object, return `true`.
6. Go to step 3.

15.2.4.7 Object.prototype.propertyIsEnumerable (V)

When the `propertyIsEnumerable` method is called with argument `V`, the following steps are taken:

1. Let `O` be the result of calling `ToObject` passing the `this` object as the argument.
2. Call `ToString(V)`.
3. Call the `[[GetOwnProperty]]` internal method of `O` passing `Result(2)` as the argument.
4. If `Result(3)` is `undefined`, return `false`.
5. Return the value of `Result(3).[[Enumerable]]`.

**NOTE**
This method does not consider objects in the prototype chain.

15.2.5 Properties of Object Instances

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

15.3 Function Objects

15.3.1 The Function Constructor Called as a Function

When `Function` is called as a function rather than as a constructor, it creates and initialises a new Function object. Thus the function call `Function(...)` is equivalent to the object creation expression `new Function(...)` with the same arguments.

15.3.1.1 Function (p1, p2, ..., pn, body)

When the `Function` function is called with some arguments `p1, p2, ..., pn, body` (where `n` might be 0, that is, there are no “p” arguments, and where `body` might also not be provided), the following steps are taken:

1. Create and return a new Function object as if the standard built-in constructor `Function` was used in a new expression with the same arguments (15.3.2.1).

15.3.2 The Function Constructor

When `Function` is called as part of a `new` expression, it is a constructor: it initialises the newly created object.

15.3.2.1 `new Function(p1, p2, ..., pn, body)

The last argument specifies the body (executable code) of a function; any preceding arguments specify formal parameters.
When the `Function` constructor is called with some arguments `p1`, `p2`, …, `pn`, `body` (where `n` might be 0, that is, there are no “`p`” arguments, and where `body` might also not be provided), the following steps are taken:

1. Let `P` be the empty string.
2. If no arguments were given, let `body` be the empty string and go to step 13.
3. If one argument was given, let `body` be that argument and go to step 13.
4. Let `Result(4)` be the first argument.
5. Let `P` be `ToString(Result(4))`.
6. Let `k` be 2.
7. If `k` equals the number of arguments, let `body` be the `k`th argument and go to step 13.
8. Let `Result(8)` be the `k`th argument.
9. Call `ToString(Result(8))`.
10. Let `P` be the result of concatenating the previous value of `P`, the string “, “ (a comma), and `Result(9)`.
11. Increase `k` by 1.
13. Call `ToString(body)`.
14. If `P` is not parsable as a `FormalParameterList opt` then throw a `SyntaxError` exception.
15. If `body` is not parsable as `FunctionBody` then throw a `SyntaxError` exception.
16. Create a new `Function` object as specified in 13.2 with parameters specified by parsing `P` as a `FormalParameterList opt` and body specified by parsing `body` as a `FunctionBody`. Pass in the `Global Environment` as the `Scope` parameter and the empty string as `Name`.
17. Return `Result(16)`.

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

**NOTE**

It is permissible but not necessary to have one argument for each formal parameter to be specified. For example, all three of the following expressions produce the same result:

```javascript
new Function("a", "b", "c", "return a+b+c")
new Function("a, b, c", "return a+b+c")
new Function("a,b", "c", "return a+b+c")
```

### 15.3.3 Properties of the Function Constructor

The value of the internal `[[Prototype]]` property of the `Function` constructor is the `Function` prototype object (15.3.4).

The value of the internal `[[Extensible]]` property of the `Function` constructor is `true`. The `Function` constructor has the following properties:

#### 15.3.3.1 `Function.prototype`

The initial value of `Function.prototype` is the `Function` prototype object (15.3.4). This property has the attributes `{[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`.

#### 15.3.3.2 `Function.length`

This is a data property with an initial value of 1. This property has the attributes `{[[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}`.

#### 15.3.4 Properties of the Function Prototype Object

The `Function prototype` object is itself a `Function` object (its `[[Class]]` is "`Function"") that, when invoked, accepts any arguments and returns `undefined`.

The value of the internal `[[Prototype]]` property of the `Function prototype` object is the `Object prototype` object (15.3.2.1). The initial value of the internal `[[Extensible]]` property of the `Function prototype` object is `true`.

15 December 2008
It is a function with an “empty body”; if it is invoked, it merely returns `undefined`.
The Function prototype object does not have a `valueOf` property of its own; however, it inherits the `valueOf` property from the Object prototype `Object`.

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

15.3.4.2 `Function.prototype.toString ( )`
An implementation-dependent representation of the function is returned. This representation has the syntax of a `FunctionDeclaration`. Note in particular that the use and placement of white space, line terminators, and semicolons within the representation string is implementation-dependent.
The `toString` function is not generic; it throws a `TypeError` exception if its this value is not a Function object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.3.4.3 `Function.prototype.apply (thisArg, argArray)`
The `apply` method takes two arguments, `thisArg` and `argArray`, and performs a function call using the `[[Call]]` property of the object. If the object does not have a `[[Call]]` property, a `TypeError` exception is thrown.

The called function is passed `thisArg` as the this value.

If `argArray` is null or `undefined`, the called function is passed no arguments. Otherwise, if `argArray` is neither an array nor an arguments object (see 10.3.2), a `TypeError` exception is thrown. If `argArray` is either an array or an arguments object, the function is passed the (ToUint32(`argArray.length`)) arguments `argArray[0], argArray[1], ..., argArray[ToUint32(`argArray.length`)-1].`

The `length` property of the `apply` method is 2.

15.3.4.4 `Function.prototype.call (thisArg [, arg1 [, arg2, ... ]])`
The `call` method takes one or more arguments, `thisArg` and (optionally) `arg1`, `arg2` etc, and performs a function call using the `[[Call]]` property of the object. If the object does not have a `[[Call]]` property, a `TypeError` exception is thrown. The called function is passed `arg1`, `arg2`, etc, as the arguments.

The called function is passed `thisArg` as the this value.

1. The `length` property of the `call` method is 1.

15.3.4.5 `Function.prototype.bind (thisArg [, arg1 [, arg2, ... ]])`
The `bind` method takes one or more arguments, `thisArg` and (optionally) `arg1`, `arg2`, etc, and returns a new function object by performing the following steps:

1. Let `T` be `thisArg`.
2. Let `G` be the this object.
3. If IsCallable(`G`) is false, throw a `TypeError` exception.
4. Let `A` be a new (possibly empty) internal list of all of the argument values provided after `thisArg` (`arg1`, `arg2` etc). in order.
5. Create a new native ECMAScript object and let `F` be that object.
6. Set the `[[Class]]` property of `F` to "`Function".
7. Set the `[[Prototype]]` property of `F` to the standard built-in Function prototype object as specified in 15.3.3.1.
8. Set the `[[Call]]` property of `F` as described in 15.3.4.5.1.
9. Set the `[[Construct]]` property of `F` as described in 15.3.4.5.2.
10. Set the `[[Scope]]` property of `F` has has no observable effect, and so can be ignored.
11. If the `[[Class]]` property of `G` is "`Function"", then
    - a. Get the `length` property of `G`.
    - b. Let `L` be `Result(11a)` minus the `length` of `A`.
    - c. Set the `length` property of `F` to either 0 or `L`, whichever is larger.
12. Else set the `length` property of `F` to 0.
13. The `length` property of `F` is given attributes as specified in 15.3.5.1.
14. Set the `[[Extensible]]` property of `F` to `true`.

15 December 2008
15. If the \([\text{Class}]\) property of \(G\) is "Function", then
   \begin{enumerate}
   \item Get the \texttt{prototype} property of \(G\).
   \item Set the \texttt{prototype} property of \(F\) to Result(15a).
   \end{enumerate}
16. Else
   \begin{enumerate}
   \item Create a new object as would be constructed by the expression \texttt{new Object()} where \texttt{Object} is the standard built-in constructor with that name.
   \item Set the \texttt{constructor} property of Result(16a) to \(F\). This property has attributes \{
   \texttt{[Writable]}: true, \texttt{[Enumerable]}: false, \texttt{[Configurable]}: true \}.
   \item Set the \texttt{prototype} property of \(F\) to Result(16a).
   \end{enumerate}
17. The \texttt{prototype} property of \(F\) is given attributes as specified in 15.3.5.2.
18. Return \(F\).

15.3.4.5.1 [[Call]]

When executed with zero or more arguments, \(F\) uses the values of \(T\), \(G\) and \(A\) that were associated with it at its creation, and the following steps are taken:
   \begin{enumerate}
   \item Let \(\text{Args}\) be a new internal list containing the same values as the list \(A\) in the same order followed by the argument list passed to \(F\) in the same order.
   \item Invoke the \texttt{[Call]} method of \(F\) providing \(T\) as the \texttt{this} value and providing \(\text{Args}\) as the arguments.
   \item Return Result(3).
   \end{enumerate}

15.3.4.5.2 [[Construct]]

When executed with zero or more arguments, \(F\) uses the values of \(G\) and \(A\) that were associated with it at its creation, and the following steps are taken:
   \begin{enumerate}
   \item If \(G\) has no \texttt{[Construct]} method, a \texttt{TypeError} exception is thrown.
   \item Let \(\text{Args}\) be a new internal list containing the same values as the list \(A\) in the same order followed by the argument list passed to \(F\) in the same order.
   \item Invoke the \texttt{[Construct]} method of \(F\) providing \texttt{undefined} as the \texttt{this} value and providing \(\text{Args}\) as the arguments.
   \item Return Result(3).
   \end{enumerate}

15.3.5 Properties of Function Instances

In addition to the required internal properties, every function instance has a \texttt{[Call]} property, a \texttt{[Construct]} property and a \texttt{[Scope]} property (see 8.6.2 and 13.2). The value of the \texttt{[Class]} property is "Function".

15.3.5.1 \texttt{length}

The value of the \texttt{length} property is an integer that indicates the "typical" number of arguments expected by the function. However, the language permits the function to be invoked with some other number of arguments. The behaviour of a function when invoked on a number of arguments other than the number specified by its \texttt{length} property depends on the function. This property has the attributes \{
   \texttt{[Writable]}: false, \texttt{[Enumerable]}: false, \texttt{[Configurable]}: false \}.

15.3.5.2 \texttt{prototype}

The value of the \texttt{prototype} property is used to initialise the internal \texttt{[Prototype]} property of a newly created object before the Function object is invoked as a constructor for that newly created object. This property has the attribute \{
   \texttt{[Writable]}: true, \texttt{[Enumerable]}: false, \texttt{[Configurable]}: false \}.

15.3.5.3 \texttt{[HasInstance]} (\(V\))

Assume \(F\) is a Function object.

When the \texttt{[HasInstance]} method of \(F\) is called with value \(V\), the following steps are taken:
   \begin{enumerate}
   \item If \(V\) is not an object, return \texttt{false}.
   \item Call the \texttt{[Get]} method of \(F\) with property name "\texttt{prototype}".
   \item Let \(O\) be Result(2).
   \item If \(O\) is not an object, throw a \texttt{TypeError} exception.
   \item Let \(V\) be the value of the \texttt{[Prototype]} property of \(V\).
   \item If \(V\) is \texttt{null}, return \texttt{false}.
   \end{enumerate}
7. If \( O \) and \( V \) refer to the same object, return \( \text{true} \).
8. Go to step 5.

15.3.5.4 \text{name}

The value of the \text{name} property is the name of the function, or an empty string if the function is anonymous. This property has the attributes { [[Writable]]: \text{false}, [[Enumerable]]: \text{true}, [[Configurable]]: \text{false} }.

15.4 \text{Array Objects}

Array objects give special treatment to a certain class of property names. A property name \( P \) (in the form of a string value) is an \text{array index} if and only if \text{ToString(ToUint32}(P)) is equal to \( P \) and ToUint32(\( P \)) is not equal to \( 2^{32} - 1 \). Every Array object has a \text{length} property whose value is always a nonnegative integer less than \( 2^{32} \). The value of the \text{length} property is numerically greater than the name of every property whose name is an array index; whenever a property of an Array object is created or changed, other properties are adjusted as necessary to maintain this invariant. Specifically, whenever a property is added whose name is an array index, the \text{length} property is changed, if necessary, to be one more than the numeric value of that array index; and whenever the \text{length} property is changed, every property whose name is an array index whose value is not smaller than the new length is automatically deleted. This constraint applies only to properties of the Array object itself and is unaffected by \text{length} or array index properties that may be inherited from its prototype.

15.4.1 \text{The Array Constructor Called as a Function}

When \text{Array} is called as a function rather than as a constructor, it creates and initialises a new Array object. Thus the function call \text{Array(...)} is equivalent to the object creation expression \text{new Array(...)} with the same arguments.

15.4.2 \text{The Array Constructor}

When \text{Array} is called as part of a \text{new} expression, it is a constructor: it initialises the newly created object.

15.4.2.1 \text{new Array ([ item0 [, item1 [, ... ]] ])}

This description applies if and only if the Array constructor is given no arguments or at least two arguments.

The [[Prototype]] property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of \text{Array.prototype} (15.4.3.1).

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

The [[Extensible]] property of the newly constructed object is set to \text{true}.

The \text{length} property of the newly constructed object is set to the number of arguments.

The 0 property of the newly constructed object is set to \text{item0} (if supplied); the 1 property of the newly constructed object is set to \text{item1} (if supplied); and, in general, for as many arguments as there are, the \( k \) property of the newly constructed object is set to argument \( k \), where the first argument is considered to be argument number 0.

15.4.2.2 \text{new Array (len)}

The [[Prototype]] property of the newly constructed object is set to the original Array prototype object, the one that is the initial value of \text{Array.prototype} (15.4.3.1). The [[Class]] property of the newly constructed object is set to "Array". The [[Extensible]] property of the newly constructed object is set to \text{true}.
If the argument `len` is a Number and `ToUint32(len)` is equal to `len`, then the `length` property of the newly constructed object is set to `ToUint32(len)`. If the argument `len` is a Number and `ToUint32(len)` is not equal to `len`, a `RangeError` exception is thrown.

If the argument `len` is not a Number, then the `length` property of the newly constructed object is set to `1` and the `0` property of the newly constructed object is set to `len`.

15.4.3 Properties of the Array Constructor

The value of the internal `[[Prototype]]` property of the Array constructor is the Function prototype object (15.3.4).

Besides the internal properties and the `length` property (whose value is `1`), the Array constructor has the following properties:

15.4.3.1 `Array.prototype`

The initial value of `Array.prototype` is the Array prototype object (15.4.4).

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.4.3.2 `Array.isArray (arg)`

The `isArray` function takes one argument `arg`, and returns the Boolean value `true` if the argument is an object whose `[[Class]]` internal property has the string value "Array", or whose `[[Prototype]]` internal property has the value of the original Array prototype; otherwise it return `false`.

15.4.4 Properties of the Array Prototype Object

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

The Array prototype object is itself an array; its `[[Class]]` is "Array", and it has a `length` property (whose initial value is `+0`) and the special internal `[[ThrowingPut]]` method described in 15.4.5.1.

In following descriptions of functions that are properties of the Array prototype object, the phrase "this object" refers to the object that is the `this` value for the invocation of the function. It is permitted for the `this` to be an object for which the value of the internal `[[Class]]` property is not "Array".

**NOTE**

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

15.4.4.1 `Array.prototype.constructor`

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

15.4.4.2 `Array.prototype.toString ( )`

The result of calling this function is the same as if the standard built-in method `Array.prototype.join` were invoked for this object with no argument.

The `toString` function is not generic; it throws a `TypeError` exception if its `this` value is not an Array object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.4.4.3 `Array.prototype.toLocaleString ( )`

The elements of the array are converted to strings using their `toLocaleString` methods, and these strings are then concatenated, separated by occurrences of a separator string that has been derived in an implementation-defined locale-specific way. The result of calling this function is intended to be analogous to the result of `toString`, except that the result of this function is intended to be locale-specific.

The result is calculated as follows:

1. Call the `[[Get]]` method of this object with argument "`length`".
2. Call `ToUint32(Result(1))`.
3. Let `separator` be the list-separator string appropriate for the host environment’s current locale (this is derived in an implementation-defined way).
4. Call `ToString(separator)`.
5. If Result(2) is zero, return the empty string.
6. Call the [[Get]] method of this object with argument "0".
7. If Result(6) is undefined or null, use the empty string; otherwise, call
   ToObject(Result(6)).toLocaleString().
8. Let R be Result(7).
9. Let k be 2.
10. If k equals Result(2), return R.
11. Let S be a string value produced by concatenating R and Result(4).
12. Call the [[Get]] method of this object with argument ToString(k).
13. If Result(12) is undefined or null, use the empty string; otherwise, call
   ToObject(Result(12)).toLocaleString().
14. Let R be a string value produced by concatenating S and Result(13).
15. Increase k by 1.

The toLocaleString function is not generic; it throws a TypeError exception if its this value is
not an Array object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

NOTE
The first parameter to this function is likely to be used in a future version of this standard; it is
recommended that implementations do not use this parameter position for anything else.

15.4.4.4 Array.prototype.concat ( [ item1 , item2 , ... ] )

When the concat method is called with zero or more arguments item1, item2, etc., it returns an array
containing the array elements of the object followed by the array elements of each argument in order.
The following steps are taken:
1. Let A be a new array created as if by the expression new Array() where Array is the standard
   built-in constructor with that name.
2. Let n be 0.
3. Let E be this object.
4. If E is not an Array object, go to step 16.
5. Let k be 0.
6. Call the [[Get]] method of E with argument "length".
7. If k equals Result(6) go to step 19.
8. Call ToString(k).
9. If E has a property named by Result(8), go to step 10, but if E has no property named by
   Result(8), go to step 13.
10. Call ToString(n).
11. Call the [[Get]] method of E with argument Result(8).
12. Call the [[Put]] method of A with arguments Result(10) and Result(11).
13. Increase n by 1.
15. Go to step 7.
16. Call ToString(n).
17. Call the [[Put]] method of A with arguments Result(16) and E.
18. Increase n by 1.
19. Get the next argument in the argument list; if there are no more arguments, go to step 22.
20. Let E be Result(19).
22. Call the [[Put]] method of A with arguments "length" and n.
23. Return A.

The length property of the concat method is 1.

NOTE
The concat function is intentionally generic; it does not require that its this value be an Array
object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the
concat function can be applied successfully to a host object is implementation-dependent.
15.4.4.5 Array.prototype.join (separator)

The elements of the array are converted to strings, and these strings are then concatenated, separated by occurrences of the separator. If no separator is provided, a single comma is used as the separator.

The `join` method takes one argument, `separator`, and performs the following steps:

1. Call the `[[Get]]` method of this object with argument "length".
2. Call ToUint32(Result(1)).
3. If `separator` is `undefined`, let `separator` be the single-character string ",".
4. Call ToString(separator).
5. If Result(2) is zero, return the empty string.
6. Call the `[[Get]]` method of this object with argument "0".
7. If Result(6) is `undefined` or `null`, use the empty string; otherwise, call ToString(Result(6)).
8. Let `R` be Result(7).
9. Let `k` be 1.
10. If `k` equals Result(2), return `R`.
11. Let `S` be a string value produced by concatenating `R` and Result(4).
12. Call the `[[Put]]` method of this object with argument ToString(`k`).
13. If Result(12) is `undefined` or `null`, use the empty string; otherwise, call ToString(Result(12)).
14. Let `R` be a string value produced by concatenating `S` and Result(13).
15. Increase `k` by 1.

The `length` property of the `join` method is 1.

*NOTE*
The `join` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the `join` function can be applied successfully to a host object is implementation-dependent.

15.4.4.6 Array.prototype.pop ( )

The last element of the array is removed from the array and returned.

1. Call the `[[Get]]` method of this object with argument "length".
2. Call ToUint32(Result(1)).
3. If Result(2) is not zero, go to step 6.
4. Call the `[[Put]]` method of this object with arguments ToString(Result(2)) and Result(3).
5. Return `undefined`.
6. Call ToString(Result(2)–1).
7. Call the `[[Get]]` method of this object with argument Result(6).
8. Call the `[[Delete]]` method of this object with argument Result(6).
9. Call the `[[Put]]` method of this object with arguments "length" and (Result(2)–1).
10. Return Result(7).

*NOTE*
The `pop` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `pop` function can be applied successfully to a host object is implementation-dependent.

15.4.4.7 Array.prototype.push ([ item1 [, item2 [, ... ] ] ])

The arguments are appended to the end of the array, in the order in which they appear. The new length of the array is returned as the result of the call.

When the `push` method is called with zero or more arguments `item1`, `item2`, etc., the following steps are taken:

1. Call the `[[Get]]` method of this object with argument "length".
2. Let `n` be the result of calling ToUint32(Result(1)).
3. Get the next argument in the argument list; if there are no more arguments, go to step 7.
4. Call the `[[Put]]` method of this object with arguments ToString(`n`) and Result(3).
5. Increase `n` by 1.
6. Go to step 3.
7. Call the [[Put]] method of this object with arguments "length" and n.
8. Return n.

The **length** property of the **push** method is 1.

**NOTE**
The **push** function is intentionally generic; it does not require that its **this** value be an **Array** object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **push** function can be applied successfully to a host object is implementation-dependent.

### 15.4.4.8 Array.prototype.reverse ( )
The elements of the array are rearranged so as to reverse their order. The object is returned as the result of the call.
1. Call the [[Get]] method of this object with argument "length".
2. Call ToInt32(Result(1)).
3. Compute floor(Result(2)/2).
4. Let k be 0.
5. If k equals Result(3), return this object.
6. Compute Result(2)-k-1.
7. Call ToString(k).
8. Call ToString(Result(6)).
9. Call the [[Get]] method of this object with argument Result(7).
10. Call the [[Get]] method of this object with argument Result(8).
11. If this object does not have a property named by Result(8), go to step 19.
12. If this object does not have a property named by Result(7), go to step 16.
13. Call the [[Put]] method of this object with arguments Result(7) and Result(10).
14. Call the [[Put]] method of this object with arguments Result(8) and Result(9).
15. Go to step 25.
16. Call the [[Put]] method of this object with arguments Result(7) and Result(10).
17. Call the [[Delete]] method on this object, providing Result(8) as the name of the property to delete.
18. Go to step 25.
19. If this object does not have a property named by Result(7), go to step 23.
20. Call the [[Delete]] method on this object, providing Result(7) as the name of the property to delete.
21. Call the [[Put]] method of this object with arguments Result(8) and Result(9).
22. Go to step 25.
23. Call the [[Delete]] method on this object, providing Result(7) as the name of the property to delete.
24. Call the [[Delete]] method on this object, providing Result(8) as the name of the property to delete.
25. Increase k by 1.
26. Go to step 5.

**NOTE**
The **reverse** function is intentionally generic; it does not require that its **this** value be an **Array** object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the **reverse** function can be applied successfully to a host object is implementation-dependent.

### 15.4.4.9 Array.prototype.shift ( )
The first element of the array is removed from the array and returned.
1. Call the [[Get]] method of this object with argument "length".
2. Call ToInt32(Result(1)).
3. If Result(2) is not zero, go to step 6.
4. Call the [[Put]] method of this object with arguments "length" and Result(2).
5. Return **undefined**.
6. Call the [[Get]] method of this object with argument 0.
7. Let \(k\) be 1.
8. If \(k\) equals Result(2), go to step 18.
9. Call ToString(\(k\)).
10. Call ToString(\(k-1\)).
11. If this object has a property named by Result(9), go to step 12; but if this object has no property
    named by Result(9), then go to step 15.
12. Call the [[Get]] method of this object with argument Result(9).
13. Call the [[Put]] method of this object with arguments Result(10) and Result(12).
14. Go to step 16.
15. Call the [[Delete]] method of this object with argument Result(10).
16. Increase \(k\) by 1.
17. Go to step 8.
18. Call the [[Delete]] method of this object with argument ToString(Result(2)–1).
19. Call the [[Put]] method of this object with arguments "length" and (Result(2)–1).
20. Return Result(6).

**NOTE**
The shift function is intentionally generic; it does not require that its this value be an Array object.
Therefore it can be transferred to other kinds of objects for use as a method. Whether the shift
function can be applied successfully to a host object is implementation-dependent.

15.4.4.10 Array.prototype.slice (start, end)
The slice method takes two arguments, start and end, and returns an array containing the elements
of the array from element start up to, but not including, element end (or through the end of the array if
end is undefined). If start is negative, it is treated as (length+start) where length is the length of the array.
If end is negative, it is treated as (length+end) where length is the length of the array. The following steps are taken:

1. Let \(A\) be a new array created as if by the expression `new Array()` where Array is the
   standard built-in constructor with that name.
2. Call the [[Get]] method of this object with argument "length".
3. Call ToUint32(Result(2)).
4. Call ToInteger(start).
5. If Result(4) is negative, use max((Result(3)+Result(4)),0); else use min(Result(4),Result(3)).
6. Let \(k\) be Result(5).
7. If end is undefined, use Result(3); else use ToInteger(end).
8. If Result(7) is negative, use max((Result(3)+Result(7)),0); else use min(Result(7),Result(3)).
9. Let \(n\) be 0.
10. If \(k\) is greater than or equal to Result(8), go to step 19.
11. Call ToString(\(k\)).
12. If this object has a property named by Result(11), go to step 13; but if this object has no property
    named by Result(11), then go to step 16.
13. Call ToString(n).
14. Call the [[Get]] method of this object with argument Result(11).
15. Call the [[Put]] method of \(A\) with arguments Result(13) and Result(14).
16. Increase \(k\) by 1.
17. Increase \(n\) by 1.
18. Go to step 10.
19. Call the [[Put]] method of \(A\) with arguments "length" and \(n\).
20. Return \(A\).

The length property of the slice method is 2.

**NOTE**
The slice function is intentionally generic; it does not require that its this value be an Array object.
Therefore it can be transferred to other kinds of objects for use as a method. Whether the slice
function can be applied successfully to a host object is implementation-dependent.
15.4.4.11 Array.prototype.sort (comparefn)

The elements of this array are sorted. The sort is not necessarily stable (that is, elements that compare equal do not necessarily remain in their original order). If comparefn is not undefined, it should be a function that accepts two arguments x and y and returns a negative value if \( x < y \), zero if \( x = y \), or a positive value if \( x > y \).

If comparefn is not undefined and is not a consistent comparison function for the elements of this array (see below), the behaviour of sort is implementation-defined. Let \( len = \text{ToUint32}(\text{this.length}) \). If there exist integers \( i \) and \( j \) and an object \( P \) such that all of the conditions below are satisfied then the behaviour of sort is implementation-defined:

\[
0 \leq i < len \\
0 \leq j < len \\
\text{this} \text{ does not have a property with name ToString}(i) \\
P \text{ is obtained by following one or more \text{[Prototype]} properties starting at this} \\
P \text{ has a property with name ToString}(j)
\]

Otherwise the following steps are taken.

1. Let \( obj \) be this object.
2. Let \( \text{getLen} \) be the result of calling the \text{[Get]} internal method of \( obj \) with argument "length".
3. Let \( len = \text{ToUint32}(\text{getLen}) \).
4. Throw a TypeError exception if any of the following conditions are true
   a. The \text{[Extensible]} internal property of \( obj \) is false.
   b. Any property of \( obj \) whose property name is a nonnegative integer less than \( len \) is a data property whose \text{[Writable]} attribute is false.
5. If any property of \( obj \) whose property name is a nonnegative integer less than \( len \) is an accessor property then the behaviour of sort is implementation-defined and the following steps may not occur.
6. Perform an implementation-dependent sequence of calls to the \text{[Get]} , \text{[Put]} , and \text{[Delete]} methods of this object and to SortCompare (described below), where the first argument for each call to \text{[Get]} , \text{[Put]} , or \text{[Delete]} is a nonnegative integer less than \( len \) and where the arguments for calls to SortCompare are results of previous calls to the \text{[Get]} method.
7. Return this object.

The returned object must have the following two properties.

There must be some mathematical permutation \( \pi \) of the nonnegative integers less than Result(2), such that for every nonnegative integer \( j \) less than Result(2), if property \( \text{old}[j] \) existed, then \( \text{new}[\pi(j)] \) is exactly the same value as \( \text{old}[j] \). But if property \( \text{old}[j] \) did not exist, then \( \text{new}[\pi(j)] \) does not exist.

Then for all nonnegative integers \( j \) and \( k \), each less than Result(2), if \( \text{SortCompare}(j,k) < 0 \) (see SortCompare below), then \( \pi(j) < \pi(k) \).

Here the notation \( \text{old}[j] \) is used to refer to the hypothetical result of calling the \text{[Get]} method of this object with argument \( j \) before this function is executed, and the notation \( \text{new}[j] \) to refer to the hypothetical result of calling the \text{[Get]} method of this object with argument \( j \) after this function has been executed.

A function comparefn is a consistent comparison function for a set of values \( S \) if all of the requirements below are met for all values \( a \), \( b \), and \( c \) (possibly the same value) in the set \( S \): The notation \( a \prec_S b \) means \( \text{comparefn}(a,b) < 0 \); \( a \prec_S b \) means \( \text{comparefn}(a,b) = 0 \) (of either sign); and \( a \succ_S b \) means \( \text{comparefn}(a,b) > 0 \).

Calling comparefn\((a,b)\) always returns the same value \( v \) when given a specific pair of values \( a \) and \( b \) as its two arguments. Furthermore, \( v \) has type Number, and \( v \) is not NaN. Note that this implies that exactly one of \( a \prec_S b \), \( a \succ_S b \), and \( a \succ_S b \) will be true for a given pair of \( a \) and \( b \).

\( a \prec_S a \) (reflexivity)
\( a \succ_S a \) (symmetry)

15 December 2008
If $a \preceq b$ and $b \preceq c$, then $a \preceq c$  
(transitivity of $\preceq$)

If $a \preceq b$ and $b < c$, then $a < c$  
(transitivity of $\preceq$

If $a \preceq b$ and $b \succeq c$, then $a \succeq c$  
(transitivity of $\succeq$)

NOTE
The above conditions are necessary and sufficient to ensure that comparefn divides the set $S$ into equivalence classes and that these equivalence classes are totally ordered.

When the SortCompare abstract operator is called with two arguments $j$ and $k$, the following steps are taken:

1. Let $jString$ be $toString(j)$.
2. Let $kString$ be $toString(k)$.
3. If this object does not have a property named by $jString$, and this object does not have a property named by $kString$, return $+0$.
4. If this object does not have a property named by $jString$, return $1$.
5. If this object does not have a property named by $kString$, return $-1$.
6. Let $x$ be the result of calling the $[[Get]]$ internal method of this object with argument $jString$.
7. Let $y$ be the result of calling the $[[Get]]$ internal method of this object with argument $kString$.
8. If $x$ and $y$ are both undefined, return $+0$.
9. If $x$ is undefined, return $1$.
10. If $y$ is undefined, return $-1$.
11. If the argument comparefn is not undefined, then:
   a. Return the result of calling the $[[Call]]$ internal method of comparefn passing undefined as the this value and with arguments $x$ and $y$.
12. Let $xString$ be $toString(x)$.
13. Let $yString$ be $toString(y)$.
14. If $xString \lt yString$, return $-1$.
15. If $xString \gt yString$, return $1$.
16. Return $+0$.

NOTE
Because non-existent property values always compare greater than undefined property values, and undefined always compares greater than any other value, undefined property values always sort to the end of the result, followed by non-existent property values.

NOTE 2
The sort function is intentionally generic; it does not require that its this value be an Array object. Therefore, it can be transferred to other kinds of objects for use as a method. Whether the sort function can be applied successfully to a host object is implementation-dependent.

15.4.4.12 Array.prototype.splice (start, deleteCount [ , item1 [ , item2 [ , ... [ ] ] ] ] )

When the splice method is called with two or more arguments start, deleteCount and (optionally) item1, item2, etc., the deleteCount elements of the array starting at array index start are replaced by the arguments item1, item2, etc. The following steps are taken:

1. Let $A$ be a new array created as if by the expression new Array{} where Array is the standard built-in constructor with that name.
2. Call the $[[Get]]$ method of this object with argument "length".
3. Call ToUint32(Result(2)).
4. Call ToInteger(start).
5. If Result(4) is negative, use max(Result(3)+Result(4),0); else use min(Result(4),Result(3)).
6. Compute min(max(ToInteger(deleteCount),0),Result(3)-Result(5)).
7. Let $k$ be 0.
8. If $k$ equals Result(6), go to step 16.
9. Call ToString(Result(5)+$k$).
10. If this object has a property named by Result(9), go to step 11; but if this object has no property named by Result(9), then go to step 14.
11. Call ToString($k$).
12. Call the $[[Call]]$ method of this object with argument Result(9).
13. Call the [[Put]] method of A with arguments Result(11) and Result(12).
15. Go to step 8.
16. Call the [[Put]] method of A with arguments “length” and Result(6).
17. Compute the number of additional arguments item1, item2, etc.
18. If Result(17) is equal to Result(6), go to step 48.
19. If Result(17) is greater than Result(6), go to step 37.
20. Let k be Result(5).
21. If k is equal to (Result(3)−Result(6)), go to step 31.
22. Call ToString(k+Result(6)).
23. Call ToString(k+Result(17)).
24. If this object has a property named by Result(22), go to step 25; but if this object has no property named by Result(22), then go to step 28.
25. Call the [[Get]] method of this object with argument Result(22).
26. Call the [[Put]] method of this object with arguments Result(23) and Result(25).
27. Go to step 29.
28. Call the [[Delete]] method of this object with argument Result(23).
29. Increase k by 1.
30. Go to step 21.
31. Let k be Result(3).
32. If k is equal to (Result(3)−Result(6)+Result(17)), go to step 48.
33. Call ToString(k−1).
34. Call the [[Delete]] method of this object with argument Result(33).
35. Decrease k by 1.
36. Go to step 32.
37. Let k be (Result(3)−Result(6)).
38. If k is equal to Result(5), go to step 48.
39. Call ToString(k+Result(6)−1).
40. Call ToString(k+Result(17)−1).
41. If this object has a property named by Result(39), go to step 42; but if this object has no property named by Result(39), then go to step 45.
42. Call the [[Get]] method of this object with argument Result(39).
43. Call the [[Put]] method of this object with arguments Result(40) and Result(42).
44. Go to step 46.
45. Call the [[Delete]] method of this object with argument Result(40).
46. Decrease k by 1.
47. Go to step 38.
48. Let k be Result(5).
49. Get the next argument in the part of the argument list that starts with item1; if there are no more arguments, go to step 53.
50. Call the [[Put]] method of this object with arguments ToString(k) and Result(49).
51. Increase k by 1.
52. Go to step 49.
53. Call the [[Put]] method of this object with arguments “length” and (Result(3)−Result(6)+Result(17)).
54. Return A.

The length property of the splice method is 2.

NOTE
The splice function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the splice function can be applied successfully to a host object is implementation-dependent.

15.4.4.13 Array.prototype.unshift ([ item1 [ , item2 [ , ... ] ] ] )
The arguments are prepended to the start of the array, such that their order within the array is the same as the order in which they appear in the argument list.

When the unshift method is called with zero or more arguments item1, item2, etc., the following steps are taken:

15 December 2008
1. Call the [[Get]] method of this object with argument "length".
2. Call ToUint32(1).
3. Compute the number of arguments.
4. Let $k$ be Result(2).
5. If $k$ is zero, go to step 15.
6. Call ToString($k-1$).
7. Call ToString($k$-Result(3)-1).
8. If this object has a property named by Result(6), go to step 9; but if this object has no property named by Result(6), then go to step 12.
9. Call the [[Get]] method of this object with argument Result(6).
10. Call the [[Put]] method of this object with the object of this method.
12. Call the [[Delete]] method of this object with argument Result(7).
13. Decrease $k$ by 1.
14. Go to step 5.
15. Let $k$ be 0.
16. Get the next argument in the part of the argument list that starts with item1; if there are no more arguments, go to step 21.
17. Call ToString($k$).
18. Call the [[Put]] method of this object with arguments Result(17) and Result(16).
19. Increase $k$ by 1.
20. Go to step 16.
21. Call the [[Put]] method of this object with arguments "length" and (Result(2)+Result(3)).
22. Return (Result(2)+Result(3)).

The **length** property of the **unshift** method is 1.

**NOTE**
The **unshift** function is intentionally generic: it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the **unshift** function can be applied successfully to a host object is implementation-dependent.

15.4.4.14 **Array.prototype.indexOf** ( searchElement [ , fromIndex ] )

**indexOf** compares **searchElement** to the elements of the array, in ascending order, using strict equality, and if found at one or more positions, returns the index of the first such position; otherwise, -1 is returned.

The optional second argument **fromIndex** defaults to 0 (i.e. the whole array is searched). If it is greater than or equal to the length of the array, -1 is returned, i.e. the array will not be searched. If it is negative, it is used as the offset from the end of the array to compute **fromIndex**. If the computed index is less than 0, the whole array will be searched.

When the **indexOf** method is called with one or two arguments, the following steps are taken:

1. Let $E$ be this object.
2. Call the [[Get]] method of $E$ with the argument "length".
3. Call ToUint32(2).
4. If Result(3) is 0, go to step 18.
5. Call ToInt32(fromIndex) (if **fromIndex** is **undefined** this step produces 0).
6. Let $n$ be Result(5).
7. If $n$ is greater than or equal to Result(3) go to step 18.
8. If $n$ is greater than or equal to 0, let $k$ be $n$, and go to step 11.
9. Let $k$ be Result(3) - abs(n).
10. If $k$ is less than 0, let $k$ be 0.
11. Call ToString($k$).
12. Call the [[Get]] method of $E$ with the argument Result(11).
13. Perform the comparison SameValue(searchElement, Result(12)).
14. If Result(13) is false go to step 16.
15. Return $k$.
16. Increase $k$ by 1.

Comment [pL13]: From Lars:
I am reasonably confident that this algorithm is not consistent with the one published as part of the JS1.6 spec:
http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference:Objects:Array:indexOf
In particular, note how the latter algorithm only performs the === comparison if the index is present in the array, whereas steps 12-14 simply call [[Get]] and use the result. I am also not confident that the bounds computations are equivalent.
(As it happens I'm not sure that the code on the mozilla site is 100% what we want either, in particular, unlike array methods in ES3 it does not appear to bound the length above at 2^32-1, for better or worse).
Anyway, anything that isn't essentially 100% compatible with the published 1.6 spec will also not be compatible with what's in ES4.
17. If \( k \) is less than Result(3) go to step 11.
18. Return -1.

**NOTE**
The `indexOf` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `indexOf` function can be applied successfully to a host object is implementation-dependent.

15.4.4.15 `Array.prototype.lastIndexOf (searchElement [ , fromIndex ] )`

`lastIndexOf` compares `searchElement` to the elements of the array in descending order using strict equality, and if found at one or more positions, returns the index of the last such position; otherwise, -1 is returned.

The optional second argument `fromIndex` defaults to the array's length (i.e. the whole array is searched). If it is greater than or equal to the length of the array, the whole array will be searched. If it is negative, it is used as the offset from the end of the array to compute `fromIndex`. If the computed index is less than 0, -1 is returned.

When the `lastIndexOf` method is called with one or two arguments, the following steps are taken:

1. Let \( E \) be this object.
2. Call the `[[Get]]` method of \( E \) with the argument "length".
3. Call `ToUint32(Result(2)).`
4. If Result(3) is 0 go to step 18.
5. Call `ToUint32(fromIndex)` (if `fromIndex` is undefined this step produces the same values as Result(3)).
6. Let \( n \) be Result(5).
7. If \( n \) is greater than or equal to Result(3), let \( k \) be Result(3) - 1, and go to step 11.
8. If \( n \) is greater than or equal to 0, let \( k \) be \( n \), and go to step 11.
9. Let \( k \) be Result(3) - `abs(n)`.
10. If \( k \) is less than 0 go to step 18.
11. Call `ToString(k).`
12. Call the `[[Get]]` method of \( E \) with the argument Result(11).
13. Perform the comparison `SameValue(searchElement, Result(12)).`
14. If Result(13) is false go to step 16.
15. Return \( k \).
16. Decrease \( k \) by 1.
17. If \( k \) is greater than or equal to 0 go to step 11.
18. Return -1.

**NOTE**
The `lastIndexOf` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `lastIndexOf` function can be applied successfully to a host object is implementation-dependent.

15.4.4.16 `Array.prototype.every (callbackfn [ , thisArg ] )`

`every` should be a function that accepts three arguments and returns the boolean value `true` or `false`. `every` calls the provided callback, as a function, once for each element present in the array, in ascending order, until it finds one where `callbackfn` returns `false`. If such an element is found, `every` immediately returns `false`. Otherwise, if `callbackfn` returned `true` for all elements, `every` will return `true`. `callbackfn` is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a `thisArg` parameter is provided, it will be used as the `this` value for each invocation of the callback. If it is not provided, `undefined` is used instead.

`callbackfn` is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

`every` does not mutate the array on which it is called.
The range of elements processed by `every` is set before the first call to `callbackfn`. Elements which are appended to the array after the call to `every` begins will not be visited by `callbackfn`. If existing elements of the array are changed, their value as passed to `callbackfn` will be the value at the time `every` visits them; elements that are deleted are not visited. `every` acts like the "for all" quantifier in mathematics. In particular, for an empty array, it returns `true`.

When the `every` method is called with one or two arguments, the following steps are taken:

1. Let `E` be this object.
2. Call the `[[Get]]` method of `E` with the argument "`length`".
3. Call `ToUint32(Result(2))`.
4. If `Result(3)` is `0` go to step 18.
5. If `Type(callbackfn)` is not Object, throw a `TypeError` exception.
6. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
7. Let `O` be `thisArg`.
8. Let `k` be `0`.
9. Call `ToString(k)`.
10. If `E` does not have a property named by `Result(9)`, go to step 16.
11. Call the `[[Get]]` method of `E` with argument `Result(9)`.
12. Call the `[[Call]]` method of `callbackfn` with `O` as the `this` value and arguments `Result(11)`, `k`, and `E`.
13. Call `ToBoolean(Result(12))`.
14. If `Result(13)` is `true` go to step 16.
15. Return `false`.
16. Increase `k` by `1`.
17. If `k` is less than `Result(3)` go to step 9.
18. Return `true`.

**NOTE**
The `every` function is intentionally generic; it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `every` function can be applied successfully to a host object is implementation-dependent.

15.4.4.17 `Array.prototype.some ( callbackfn [, thisArg ] )`

`callbackfn` should be a function that accepts three arguments and returns the boolean value `true` or `false`. `some` calls the callback, as a function, once for each element present in the array, in ascending order, until it finds one where `callbackfn` returns `true`. If such an element is found, `some` immediately returns `true`. Otherwise, `some` returns `false`. `callbackfn` is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a `thisArg` parameter is provided, it will be used as the `this` value for each invocation of the callback. If it is not provided, `undefined` is used instead.

`callbackfn` is called with three arguments: the value of the element, the index of the element, and the Array object being traversed.

`some` does not mutate the array on which it is called.

The range of elements processed by `some` is set before the first call to `callbackfn`. Elements that are appended to the array after the call to `some` begins will not be visited by `callbackfn`. If an existing, unvisited element of the array is changed by `callbackfn`, their value as passed to `callbackfn` will be the value at the time that `some` visits them; elements that are deleted are not visited.

When the `some` method is called with one or two arguments, the following steps are taken:

1. Let `E` be this object.
2. Call the `[[Get]]` method of `E` with the argument "`length`".
3. Call `ToUint32(Result(2))`.
4. If `Result(3)` is `0` go to step 18.
5. If `Type(callbackfn)` is not Object, throw a `TypeError` exception.
6. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
7. Let \( O \) be \( \text{thisArg} \).
8. Let \( k \) be 0.
9. Call To String(\( k \)).
10. If \( E \) does not have a property named by Result(9), go to step 16.
11. Call the \([\text{Get}]\) method of \( E \) with argument Result(9).
12. Call the \([\text{Call}]\) method of \( \text{callbackfn} \) with \( O \) as the \( \text{this} \) value and arguments Result(11), \( k \), and \( E \).
13. Call To Boolean(Result(12)).
14. If Result(13) is \( \text{false} \) go to step 16.
15. Return \( \text{true} \).
16. Increase \( k \) by 1.
17. If \( k \) is less than Result(3) go to step 9.
18. Return \( \text{false} \).

**NOTE**
The \text{some} function is intentionally generic; it does not require that its \text{this} value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the \text{some} function can be applied successfully to a host object is implementation-dependent.

**15.4.4.18 Array.prototype\text{forEach} ( callbackfn [ , thisArg ] )**
callbackfn should be a function that accepts three arguments. \text{forEach} calls the provided callback, as a function, once for each element present in the array, in ascending order. \text{callbackfn} is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a \text{thisArg} parameter is provided, it will be used as the \text{this} value for each invocation of the callback. If it is not provided, \text{undefined} is used instead.

\text{callbackfn} is called with three arguments: the value of the element, the index of the element, and the Array object being traversed. \text{forEach} does not mutate the array on which it is called.

The range of elements processed by \text{forEach} is set before the first call to \text{callbackfn}. Elements which are appended to the array after the call to \text{forEach} begins will not be visited by \text{callbackfn}. If existing elements of the array are changed, or deleted, their value as passed to callback will be the value at the time \text{forEach} visits them; elements that are deleted are not visited.

When the \text{forEach} method is called with one or two arguments, the following steps are taken:
1. Let \( E \) be this object.
2. Call the \([\text{Get}]\) method of \( E \) with the argument \"length\".
3. Call To Uint32(Result 2)).
4. If Result(3) is 0 go to step 14.
5. If Type(callbackfn) is \text{Object}, throw a \text{TypeError} exception.
6. If Is Callable(callbackfn) is \text{false}, throw a \text{TypeError} exception.
7. Let \( O \) be \text{thisArg}.
8. Let \( k \) be 0.
9. Call To String(\( k \)).
10. If \( E \) does not have a property named by Result(9), go to step 13.
11. Call the \([\text{Get}]\) method of \( E \) with argument Result(9).
12. Call the \([\text{Call}]\) method of \( \text{callbackfn} \) with \( O \) as \text{this} value and arguments Result(11), \( k \), and \( E \).
13. Increase \( k \) by 1.
14. If \( k \) is less than Result(3) go to step 9.
15. Return.

**NOTE**
The \text{forEach} function is intentionally generic; it does not require that its \text{this} value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the \text{forEach} function can be applied successfully to a host object is implementation-dependent.
15.4.4.19 `Array.prototype.map (callbackfn [ , thisArg ])

`callbackfn` should be a function that accepts three arguments. `map` calls the provided callback, as a function, once for each element in the array, in ascending order, and constructs a new array from the results. `callbackfn` is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a `thisArg` parameter is provided, it will be used as the `this` value for each invocation of the callback. If it is not provided, `undefined` is used instead.

`callbackfn` is called with three arguments: the value of the element, the index of the element, and the `Array` object being traversed.

`map` does not mutate the array on which it is called.

The range of elements processed by `map` is set before the first call to `callbackfn`. Elements which are appended to the array after the call to `map` begins will not be visited by `callbackfn`. If existing elements of the array are changed, or deleted, their value as passed to `callbackfn` will be the value at the time `map` visits them; elements that are deleted are not visited.

When the `map` method is called with one or two arguments, the following steps are taken:

1. Let `A` be a new array created as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.
2. Let `n` be 0.
3. Let `E` be this object.
4. Call the `[[Get]]` method of `E` with the argument "length".
5. Call `ToUint32(Result(4)).`  
6. If `Result(5)` is 0 go to step 20.
7. If `IsCallable(callbackfn)` is `false`, throw a `TypeError` exception.
8. Call the `[[Call]]` method of `callbackfn` with `O` as the `this` value and arguments `Result(13)`, `k`, and `E`.
9. Call `ToString(n)`.
10. Call the `[[Put]]` method of `A` with the argument `Result(14)` and `Result(15)`.
11. Increase `n` by 1.
12. Increase `k` by 1.
13. If `k` is less than `Result(5)` go to step 11.

**NOTE**
The `map` function is intentionally generic; it does not require that its `this` value be an `Array` object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `map` function can be applied successfully to a host object is implementation-dependent.

15.4.4.20 `Array.prototype.filter (callbackfn [ , thisArg ])

`callbackfn` should be a function that accepts three arguments and returns the boolean value `true` or `false`. `filter` calls the provided callback, as a function, once for each element in the array, in ascending order, and constructs a new array of all the values for which `callbackfn` returns `true`. `callbackfn` is called only for indexes of the array which have assigned values; it is not called for indexes which have been deleted or which have never been assigned values.

If a `thisArg` parameter is provided, it will be used as the `this` value for each invocation of the callback. If it is not provided, `undefined` is used instead.

`callbackfn` is called with three arguments: the value of the element, the index of the element, and the `Array` object being traversed.

`filter` does not mutate the array on which it is called.
The range of elements processed by `filter` is set before the first call to `callbackfn`. Elements which are appended to the array after the call to `filter` begins will not be visited by `callbackfn`. If existing elements of the array are changed, or deleted, their value as passed to `callbackfn` will be the value at the time `filter` visits them; elements that are deleted are not visited.

When the `filter` method is called with one or two arguments, the following steps are taken:

1. Let `A` be a new array created as if by the expression `new Array()` where `Array` is the standard built-in constructor with that name.
2. Let `n` be 0.
3. Let `E` be this object.
4. Call the `[[Get]]` method of `E` with the argument "length".
5. Call `ToUint32(Result(4))`.
6. If `Result(5)` is 0 go to step 22.
7. If `Type(callbackfn)` is not Object, throw a `TypeError` exception.
8. If `IsCallable(callbackfn)` is false, throw a `TypeError` exception.
9. Let `O` this `thisArg`.
10. Let `k` be 0.
11. Call `ToString(k)`.
12. If `E` does not have a property named by `Result(11)`, go to step 20.
13. Call the `[[Get]]` method of `E` with argument `Result(11)`.
14. Call the `[[Call]]` method of `callbackfn` with `O` as the `this` value and arguments `Result(13)`, `k`, and `E`.
15. Call `ToBoolean(Result(14))`.
16. If `Result(15)` is `false` go to step 20.
17. Call `ToString(n)`.
18. Call the `[[Put]]` method of `A` with the argument `Result(13)` and `Result(17)`.
19. Increase `n` by 1.
20. Increase `k` by 1.
21. If `k` is less than `Result(5)` go to step 11.
22. Return `A`.

**NOTE**
The `filter` function is intentionally generic: it does not require that its `this` value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the `filter` function can be applied successfully to a host object is implementation-dependent.

**15.4.4.21 Array.prototype.reduce ( callbackfn [ , initialValue ] )**

`callbackfn` should be a function that takes four arguments. `reduce` calls the callback, as a function, once for each element present in the array, in ascending order.

`callbackfn` is called with four arguments: the `previousValue` (or value from the previous call to `callbackfn`), the `currentValue` (value of the current element), the `currentIndex`, and the Array object being traversed. The first time that `callback` is called, the `previousValue` and `currentValue` can be one of two values. If an `initialValue` was provided in the call to `reduce`, then `previousValue` will be equal to `initialValue` and `currentValue` will be equal to the first value in the array. If no `initialValue` was provided, then `previousValue` will be equal to the first value in the array and `currentValue` will be equal to the second.

`reduce` does not mutate the array on which it is called.

The range of elements processed by `reduce` is set before the first call to `callbackfn`. Elements that are appended to the array after the call to `reduce` begins will not be visited by `callbackfn`. If an existing, unvisited element is changed by `callbackfn`, their value as passed to `callbackfn` will be the value at the time `reduce` visits them; elements that are deleted are not visited.

When the `reduce` method is called with one or two arguments, the following steps are taken:

1. Let `E` be this object.
2. Call the `[[Get]]` method on `E` with argument "length".
3. Call `ToUint32(Result(2))`.
4. If `Type(callbackfn)` is not Object, throw a `TypeError` exception.
5. If IsCallable(callbackfn) is false, throw a TypeError exception.
6. If Result(3) is 0 and initialValue is not supplied throw a TypeError exception.
7. Let k be 0.
8. If initial\_Value is supplied let P be initialValue and go to step 17.
9. Call ToString(k).
10. If E does not have a property named by Result(9), go to step 14.
11. Call the [[Get]] method on E with the argument Result(9).
12. Increase k by 1.
13. Let P be Result(11) and go to step 17.
15. If \( k < \) Result(3) go to step 9.
16. Throw a TypeError exception.
17. Call ToString(k).
18. If E does not have a property named by Result(17), go to step 22.
19. Call the [[Get]] method of E with the argument Result(17).
20. Call the [[Get]] method on callbackfn with null as the this value and arguments P, Result(19), k, E.
21. Let P be Result(20).
22. Increase k by 1.
23. If \( k < \) Result(3) go to step 17.
24. Return P.

NOTE
The reduce function is intentionally generic; it does not require that its this value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the reduce function can be applied successfully to a host object is implementation-dependent.

15.4.4.2 Array.prototype.reduceRight ( callbackfn [, initialValue ] )
callbackfn should be a function that takes four arguments; reduceRight calls the callback, as a function, once for each element present in the array, in descending order.
callbackfn is called with four arguments: the previous\_value (or value from the previous call to callbackfn), the current\_value (value of the current element), the current\_index, and the Array object being traversed. The first time the function is called, the previous\_value and current\_value can be one of two values. If an initial\_Value was provided in the call to reduceRight, then previous\_value will be equal to initial\_Value and current\_value will be equal to the last value in the array. If no initial\_Value was provided, then previous\_value will be equal to the last value in the array and current\_value will be equal to the second-to-last value.
reduceRight does not mutate the array on which it is called.
The range of elements processed by reduceRight is set before the first call to callbackfn. Elements that are appended to the array after the call to reduceRight begins will not be visited by callbackfn. If an existing, unvisited element is changed by callbackfn, their value as passed to callbackfn will be the value at the time reduceRight visits them; elements that are deleted are not visited.
When the reduceRight method is called with one or two arguments, the following steps are taken:
1. Let E be this object.
2. Call the [[Get]] method on E with argument "length".
3. Call ToUint32(Result(2)).
4. If Type(callbackfn) is not Object throw a TypeError exception.
5. If IsCallable(callbackfn) is false, throw a TypeError exception.
6. If Result(3) is 0 and initial\_Value is not supplied throw a TypeError exception.
7. Let k be Result(3) – 1.
8. If initial\_Value is supplied let P be initial\_Value and go to step 17.
9. Call ToString(k).
10. If E does not have a property named by Result(9), go to step 14.
11. Call the [[Get]] method on E with the argument Result(9).
12. Decrease k by 1.

15 December 2008
13. Let \( P \) be Result(11) and go to step 17.
14. Decrease \( k \) by 1.
15. If \( k \) is greater than or equal to 0 go to step 9.
16. Throw a \texttt{TypeError} exception.
17. Call \texttt{toString(k)}.
18. If \( E \) does not have a property named by Result(17), go to step 22.
19. Call the \texttt{[[Get]]} method of \( E \) with the argument Result(17).
20. Call the \texttt{[[Call]]} method on \texttt{callbackfn} with \texttt{null} as the \texttt{this} value and arguments \( P \), Result(19), \( k \), \( E \).
21. Let \( P \) be Result(20).
22. Decrease \( k \) by 1.
23. If \( k \) is greater than or equal to 0 go to step 17.
24. Return \( P \).

\textbf{NOTE}

The \texttt{reduceRight} function is intentionally generic; it does not require that its this value be an \texttt{Array} object. Therefore it can be transferred to other kinds of objects for use as a method. Whether the \texttt{reduceRight} function can be applied successfully to a host object is implementation-dependent.

### 15.4.5 Properties of Array Instances

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

\[ 15.4.5.1 \]

\texttt{[[ThrowingPut]]} ( \( P \), \( V \), \( Throw \))

Array objects use a variation of the \texttt{[[ThrowingPut]]} method used for other native ECMAScript objects (8.6.2.10).

Assume \( A \) is an Array object, \( P \) is a string, and \( Throw \) is a boolean flag.

When the \texttt{[[ThrowingPut]]} method of \( A \) is called with property \( P \), value \( V \), and Boolean flag \( Throw \), the following steps are taken:

1. Call the \texttt{[[CanPut]]} method of \( A \) with name \( P \).
2. If Result(1) is \texttt{false}, then
   a. If \( Throw \) is \texttt{true}, throw a \texttt{TypeError} exception.
   b. Else return.
3. If \( A \) doesn’t have a property with name \( P \), go to step 7.
4. If \( P \) is "length", go to step 12.
5. Set the value of property \( P \) of \( A \) to \( V \).
7. Create a property with name \( P \) of \( A \), set its value to \( V \) and give it empty attributes.
8. If \( P \) is not an array index, return.
9. If ToUint32(\( P \)) is less than the value of the length property of \( A \), then return.
10. Change (or set) the value of the length property of \( A \) to ToUint32(\( P \)+1).
11. Return.
12. Compute ToUint32(\( V \)).
13. If Result(12) is not equal to ToNumber(\( V \)), throw a RangeError exception.
14. For every integer \( k \) that is less than the value of the length property of \( A \) but not less than Result(12), if \( A \) itself has an own property (a non-inherited property) named \texttt{toString(\( k \))}, then delete that property.
15. Set the value of property \( P \) of \( A \) to Result(12).
16. Return.

\[ 15.4.5.2 \]

\texttt{length}

The \texttt{length} property of this Array object is always numerically greater than the name of every property whose name is an array index.

The \texttt{length} property has the attributes \{ \texttt{[Enumerable]}: \texttt{false}, \texttt{[Configurable]}: \texttt{false} \}.  

15 December 2008
15.5 String Objects

15.5.1 The String Constructor Called as a Function

When `String` is called as a function rather than as a constructor, it performs a type conversion.

15.5.1.1 `String ( [ value ] )`

Returns a string value (not a `String` object) computed by `ToString(value)`. If `value` is not supplied, the empty string "" is returned.

15.5.2 The String Constructor

When `String` is called as part of a `new` expression, it is a constructor: it initialises the newly created object.

15.5.2.1 `new String ( [ value ] )`

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

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

The `[[PrimitiveValue]]` property of the newly constructed object is set to `ToString(value)`, or to the empty string if `value` is not supplied.

15.5.3 Properties of the String Constructor

The value of the internal `[[Prototype]]` property of the `String` constructor is the `Function` prototype object (15.3.4).

Besides the internal properties and the `length` property (whose value is 1), the `String` constructor has the following properties:

15.5.3.1 `String.prototype`

The initial value of `String.prototype` is the `String` prototype object (15.5.4).

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.5.3.2 `String.fromCharCode ( [ char0 [ , char1 [ , ... ] ] ] )`

Returns a string value containing as many characters as the number of arguments. Each argument specifies one character of the resulting string, with the first argument specifying the first character, and so on, from left to right. An argument is converted to a character by applying the operation `ToUint16` (9.7) and regarding the resulting 16-bit integer as the code point value of a character. If no arguments are supplied, the result is the empty string.

The `length` property of the `fromCharCode` function is 1.

15.5.4 Properties of the String Prototype Object

The `String` prototype object is itself a `String` object (its `[[Class]]` is "String") whose value is an empty string.

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

15.5.4.1 `String.prototype.constructor`

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

15.5.4.2 `String.prototype.toString ( )`

Returns this string value. (Note that, for a `String` object, the `toString` method happens to return the same thing as the `valueOf` method.)

The `toString` function is not generic; it throws a `TypeError` exception if its `this` value is not a `String` object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.5.4.3 `String.prototype.valueOf ( )`

Returns this string value.

15 December 2008
The `valueOf` function is not generic; it throws a `TypeError` exception if its `this` value is not a String object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

### 15.5.4.4 String.prototype.charAt (pos)

Returns a string containing the character at position `pos` in the string resulting from converting this object to a string. If there is no character at that position, the result is the empty string. The result is a string value, not a String object.

If `pos` is a value of Number type that is an integer, then the result of `x.charAt(pos)` is equal to the result of `x.substring(pos, pos+1)`.

When the `charAt` method is called with one argument `pos`, the following steps are taken:

1. Call `ToString`, giving it the `this` value as its argument.
2. Call `ToInteger(pos)`,
3. Compute the number of characters in Result(1).
4. If Result(2) is less than 0 or is not less than Result(3), return the empty string.
5. Return a string of length 1, containing one character from Result(1), namely the character at position Result(2), where the first (leftmost) character in Result(1) is considered to be at position 0, the next one at position 1, and so on.

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

### 15.5.4.5 String.prototype.charCodeAt (pos)

Returns a number (a nonnegative integer less than $2^{16}$) representing the code point value of the character at position `pos` in the string resulting from converting this object to a string. If there is no character at that position, the result is `NaN`.

When the `charCodeAt` method is called with one argument `pos`, the following steps are taken:

1. Call `ToString`, giving it the `this` value as its argument.
2. Call `ToInteger(pos)`.
3. Compute the number of characters in Result(1).
4. If Result(2) is less than 0 or is not less than Result(3), return `NaN`.
5. Return a value of Number type, whose value is the code point value of the character at position Result(2) in the string Result(1), where the first (leftmost) character in Result(1) is considered to be at position 0, the next one at position 1, and so on.

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

### 15.5.4.6 String.prototype.concat ( [ string1 [ , string2 [ , … ] ] ] )

When the `concat` method is called with zero or more arguments `string1`, `string2`, etc., it returns a string consisting of the characters of this object (converted to a string) followed by the characters of each of `string1`, `string2`, etc. (where each argument is converted to a string). The result is a string value, not a String object. The following steps are taken:

1. Call `ToString`, giving it the `this` value as its argument.
2. Let `R` be Result(1).
3. Get the next argument in the argument list; if there are no more arguments, go to step 7.
4. Call `ToString(Result(3))`,
5. Let `R` be the string value consisting of the characters in the previous value of `R` followed by the characters `Result(4)`.
6. Go to step 3.
7. Return `R`.

The `length` property of the `concat` method is 1.

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

15.5.4.7 `String.prototype.indexOf (searchString, position)`

If `searchString` appears as a substring of the result of converting this object to a string, at one or more positions that are greater than or equal to `position`, then the index of the smallest such position is returned; otherwise, `-1` is returned. If `position` is `undefined`, `0` is assumed, so as to search all of the string.

The `indexOf` method takes two arguments, `searchString` and `position`, and performs the following steps:

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

The `length` property of the `indexOf` method is 1.

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

15.5.4.8 `String.prototype.lastIndexOf (searchString, position)`

If `searchString` appears as a substring of the result of converting this object to a string at one or more positions that are smaller than or equal to `position`, then the index of the greatest such position is returned; otherwise, `-1` is returned. If `position` is `undefined`, the length of the string value is assumed, so as to search all of the string.

The `lastIndexOf` method takes two arguments, `searchString` and `position`, and performs the following steps:

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

The `length` property of the `lastIndexOf` method is 1.

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

15.5.4.9 `String.prototype.localeCompare (that)`

When the `localeCompare` method is called with one argument `that`, it returns a number other than `NaN` that represents the result of a locale-sensitive string comparison of this object (converted to a
string) with that (converted to a string). The two strings are compared in an implementation-defined fashion. The result is intended to order strings in the sort order specified by the system default locale, and will be negative, zero, or positive, depending on whether this comes before that in the sort order, the strings are equal, or this comes after that in the sort order, respectively.

The `localeCompare` method, if considered as a function of two arguments this and that, is a consistent comparison function (as defined in 15.4.4.11) on the set of all strings. Furthermore, `localeCompare` returns 0 or –0 when comparing two strings that are considered canonically equivalent by the Unicode standard.

The actual return values are left implementation-defined to permit implementers to encode additional information in the result value, but the function is required to define a total ordering on all strings and to return 0 when comparing two strings that are considered canonically equivalent by the Unicode standard.

NOTE 1
The localeCompare method itself is not directly suitable as an argument to `Array.prototype.sort` because the latter requires a function of two arguments.

NOTE 2
This function is intended to rely on whatever language-sensitive comparison functionality is available to the ECMAScript environment from the host environment, and to compare according to the rules of the host environment's current locale. It is strongly recommended that this function treat strings that are canonically equivalent according to the Unicode standard as identical (in other words, compare the strings as if they had both been converted to Normalised Form C or D first). It is also recommended that this function not honour Unicode compatibility equivalences or decompositions.

If no language-sensitive comparison at all is available from the host environment, this function may perform a bitwise comparison.

NOTE 3
The localeCompare function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

NOTE 4
The second parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.5.4.10 String.prototype.match (regexp)
If regexp is not an object whose [[Class]] property is "RegExp", it is replaced with the result of the expression `new RegExp(regexp)`. Let string denote the result of converting the this value to a string. Then do one of the following:

If regexp.global is false: Return the result obtained by invoking `RegExp.prototype.exec` (see 15.10.6.2) on regexp with string as parameter.

If regexp.global is true: Set the regexp.lastIndex property to 0 and invoke `RegExp.prototype.exec` repeatedly until there is no match. If there is a match with an empty string (in other words, if the value of regexp.lastIndex is left unchanged), increment regexp.lastIndex by 1. Let n be the number of matches. If n=0, then the value returned is `null`; otherwise, the value returned is an array with the length property set to n and properties 0 through n-1 corresponding to the first elements of the results of all matching invocations of `RegExp.prototype.exec`.

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

15.5.4.11 String.prototype.replace (searchValue, replaceValue)
Let string denote the result of converting the this value to a string.

If searchValue is a regular expression (an object whose [[Class]] property is "RegExp"), do the following: If searchValue.global is false, then search string for the first match of the regular
expression searchValue. If searchValue global is true, then search string for all matches of the regular expression searchValue. Do the search in the same manner as in String.prototype.match, including the update of searchValue.lastIndex. Let m be the number of left capturing parentheses in searchValue (NcapturingParen as specified in 15.10.2.1).

If searchValue is not a regular expression, let searchString be ToString(searchValue) and search string for the first occurrence of searchString. Let m be 0.

If replaceValue is a function, then for each matched substring, call the function with the following m + 3 arguments. Argument 1 is the substring that matched. If searchValue is a regular expression, the next m arguments are all of the captures in the MatchResult (see 15.10.2.1). Argument m + 2 is the offset within string where the match occurred, and argument m + 3 is string. The result is a string value derived from the original input by replacing each matched substring with the corresponding return value of the function call, converted to a string if need be.

Otherwise, let newstring denote the result of converting replaceValue to a string. The result is a string value derived from the original input string by replacing each matched substring with a string derived from newstring by replacing characters in newstring by replacement text as specified in the following table. These $ replacements are done left-to-right, and, once such a replacement is performed, the new replacement text is not subject to further replacements. For example, "$1,$2".replace(/(/\$\(\d))/g, "$1$1,$2$2") returns "$1$11,$1$22". A $ in newstring that does not match any of the forms below is left as is.

<table>
<thead>
<tr>
<th>Characters</th>
<th>Replacement text</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n</td>
<td>The matched substring.</td>
</tr>
<tr>
<td>$n</td>
<td>The portion of string that precedes the matched substring.</td>
</tr>
<tr>
<td>$n</td>
<td>The portion of string that follows the matched substring.</td>
</tr>
<tr>
<td>$n</td>
<td>The nth capture, where n is a single digit 1-9 and $n is not followed by a decimal digit. If n&gt;m and the nth capture is undefined, use the empty string instead. If n&gt;m, the result is implementation-defined.</td>
</tr>
<tr>
<td>$nn</td>
<td>The nth capture, where nn is a two-digit decimal number 01-99. If nn&gt;m and the nth capture is undefined, use the empty string instead. If nn&gt;m, the result is implementation-defined.</td>
</tr>
</tbody>
</table>

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

15.5.4.12 String.prototype.search (regexp)

If regexp is not an object whose [[Class]] property is "RegExp", it is replaced with the result of the expression new RegExp(regexp). Let string denote the result of converting the this value to a string.

The value string is searched from its beginning for an occurrence of the regular expression pattern regexp. The result is a number indicating the offset within the string where the pattern matched, or −1 if there was no match.

NOTE 1
This method ignores the lastIndex and global properties of regexp. The lastIndex property of regexp is left unchanged.

NOTE 2
The search function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.
15.5.4.13 String.prototype.slice (start, end)

The slice method takes two arguments, start and end, and returns a substring of the result of converting this object to a string, starting from character position start and running to, but not including, character position end (or through the end of the string if end is undefined). If start is negative, it is treated as (sourceLength+start) where sourceLength is the length of the string. If end is negative, it is treated as (sourceLength+end) where sourceLength is the length of the string. The result is a string value, not a String object. The following steps are taken:

1. Call ToString, giving it the this value as its argument.
2. Compute the number of characters in Result(1).
3. Call ToInteger(start).
4. If end is undefined, use Result(2); else use ToInteger(end).
5. If Result(3) is negative, use max(Result(2)+Result(3),0); else use min(Result(3),Result(2)).
6. If Result(4) is negative, use max(Result(2)+Result(4),0); else use min(Result(4),Result(2)).
7. Compute max(Result(6)–Result(5),0).
8. Return a string containing Result(7) consecutive characters from Result(1) beginning with the character at position Result(5).

The length property of the slice method is 2.

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

15.5.4.14 String.prototype.split (separator, limit)

Returns an Array object into which substrings of the result of converting this object to a string have been stored. The substrings are determined by searching from left to right for occurrences of separator; these occurrences are not part of any substring in the returned array, but serve to divide up the string value. The value of separator may be a string of any length or it may be a RegExp object (i.e., an object whose [[Class]] property is "RegExp"; see 15.10).

The value of separator may be an empty string, an empty regular expression, or a regular expression that can match an empty string. In this case, separator does not match the empty substring at the beginning or end of the input string, nor does it match the empty substring at the end of the previous separator match. (For example, if separator is the empty string, the string is split up into individual characters; the length of the result array equals the length of the string, and each substring contains one character.) If separator is a regular expression, only the first match at a given position of the this string is considered, even if backtracking could yield a non-empty-substring match at that position. (For example, "ab".split(/a*?/) evaluates to the array ["a", "b"], while "ab".split(/a*/) evaluates to the array ["", "b"]).

If the this object is (or converts to) the empty string, the result depends on whether separator can match the empty string. If it can, the result array contains no elements. Otherwise, the result array contains one element, which is the empty string.

If separator is a regular expression that contains capturing parentheses, then each time separator is matched the results (including any undefined results) of the capturing parentheses are spliced into the output array. (For example, "A<B>bold</B>and<CODE>coded</CODE>".split(/<(\/)?{[^<>]+}>/) evaluates to the array ["A", undefined, "B", "bold", "/", "B", "and", undefined, "CODE", "coded", "/", "CODE", "]").

If separator is undefined, then the result array contains just one string, which is the this value (converted to a string). If limit is not undefined, then the output array is truncated so that it contains no more than limit elements.

When the split method is called, the following steps are taken:

1. Let S = ToString(this).
2. Let A be a new array created as if by the expression new Array() where Array is the standard built-in constructor with that name.
3. If limit is undefined, let lim = 2^32–1; else let lim = ToUint32(limit).
4. Let $s$ be the number of characters in $S$.
5. Let $p = 0$.
6. If separator is a RegExp object (its [[Class]] is "RegExp"), let $R = \text{separator};$ otherwise let $R = \text{ToString(separator)}$.
7. If $\text{lim} = 0$, return $A$.
8. If separator is undefined, go to step 33.
9. If $s = 0$, go to step 31.
10. Let $q = p$.
11. If $q = s$, go to step 28.
12. Call \text{SplitMatch}($R$, $S$, $q$) and let $z$ be its MatchResult result.
13. If $z$ is failure, go to step 26.
14. $z$ must be a State. Let $e$ be $z$'s endIndex and let $\text{cap}$ be $z$'s captures array.
15. If $e = p$, go to step 26.
16. Let $T$ be a string value equal to the substring of $S$ consisting of the characters at positions $p$ (inclusive) through $q$ (exclusive).
17. Call the 
\[[\text{Put}]\]
method of $A$ with arguments $A$.length and $T$.
18. If $A$.length = lim, return $A$.
19. Let $p = e$.
20. Let $i = 0$.
21. If $i$ is equal to the number of elements in $\text{cap}$, go to step 10.
22. Let $i = i + 1$.
23. Call the 
\[[\text{Put}]\]
method of $A$ with arguments $A$.length and $\text{cap}[i]$.
24. If $A$.length = lim, return $A$.
25. Go to step 21.
26. Let $q = q + 1$.
27. Go to step 11.
28. Let $T$ be a string value equal to the substring of $S$ consisting of the characters at positions $p$ (inclusive) through $q$ (exclusive).
29. Call the 
\[[\text{Put}]\]
method of $A$ with arguments $A$.length and $T$.
30. Return $A$.
31. Call \text{SplitMatch}($R$, $S$, 0) and let $z$ be its MatchResult result.
32. If $z$ is not failure, return $A$.
33. Call the 
\[[\text{Put}]\]
method of $A$ with arguments "0" and $S$.
34. Return $A$.

The abstract operation \text{SplitMatch} takes three parameters, a string $S$, an integer $q$, and a string or RegExp $R$, and performs the following in order to return a MatchResult (see 15.10.2.1):
1. If $R$ is a RegExp object (its [[Class]] is "RegExp"), go to step 8.
2. $R$ must be a string. Let $r$ be the number of characters in $R$.
3. Let $s$ be the number of characters in $S$.
4. If $q + r > s$ then return the MatchResult failure.
5. If there exists an integer $i$ between $0$ (inclusive) and $r$ (exclusive) such that the character at position $q + i$ of $S$ is different from the character at position $i$ of $R$, then return failure.
6. Let $\text{cap}$ be an empty array of captures (see 15.10.2.1).
7. Return the State ($q + r$, $\text{cap}$). (see 15.10.2.1)
8. Call the \[[\text{Match}]\] method of $R$ giving it the arguments $S$ and $q$, and return the MatchResult result.

The \text{length} property of the \text{split} method is 2.

\text{NOTE 1}

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

\text{NOTE 2}

The \text{split} method ignores the value of separator.global for separators that are RegExp objects.

\text{15.5.4.15 String.prototype.substring (start, end)}

The \text{substring} method takes two arguments, \text{start} and \text{end}, and returns a substring of the result of converting this object to a string, starting from character position \text{start} and running to, but not
including, character position end of the string (or through the end of the string is end is undefined). The result is a string value, not a String object.

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

If start is larger than end, they are swapped.

The following steps are taken:
1. Call ToString, giving it the this value as its argument.
2. Compute the number of characters in Result(1).
3. Call ToInteger(start).
4. If end is undefined, use Result(2); else use ToInteger(end).
5. Compute min(max(Result(3), 0), Result(2)).
6. Compute min(max(Result(4), 0), Result(2)).
7. Compute min(Result(5), Result(6)).
8. Compute max(Result(5), Result(6)).
9. Return a string whose length is the difference between Result(8) and Result(7), containing characters from Result(1), namely the characters with indices Result(7) through Result(8) - 1, in ascending order.

The length property of the substring method is 2.

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

15.5.4.16 String.prototype.toLowerCase ()
If this object is not already a string, it is converted to a string. The characters in that string are converted one by one to lower case. The result is a string value, not a String object.

The characters are converted one by one. The result of each conversion is the original character, unless that character has a Unicode lowercase equivalent, in which case the lowercase equivalent is used instead.

NOTE 1
The result should be derived according to the case mappings in the Unicode character database (this explicitly includes not only the UnicodeData.txt file, but also the SpecialCasings.txt file that accompanies it in Unicode 2.1.8 and later).

NOTE 2
The toLowerCase function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.17 String.prototype.toLocaleLowerCase ()
This function works exactly the same as toLowerCase except that its result is intended to yield the correct result for the host environment’s current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

NOTE 1
The toLocaleLowerCase function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

NOTE 2
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.5.4.18 String.prototype.toUpperCase ()
This function behaves in exactly the same way as String.prototype.toLowerCase, except that characters are mapped to their uppercase equivalents as specified in the Unicode Character Database.
NOTE 1
Because both `toUpperCase` and `toLowerCase` have context-sensitive behaviour, the functions are not symmetrical. In other words, `s.toUpperCase().toLowerCase()` is not necessarily equal to `s.toLowerCase()`.

NOTE 2
The `toUpperCase` function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

15.5.4.19 `String.prototype.toLocaleUpperCase ( )`
This function works exactly the same as `toUpperCase` except that its result is intended to yield the correct result for the host environment’s current locale, rather than a locale-independent result. There will only be a difference in the few cases (such as Turkish) where the rules for that language conflict with the regular Unicode case mappings.

NOTE 1
The `toLocaleUpperCase` function is intentionally generic; it does not require that its this value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

NOTE 2
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.5.4.20 `String.prototype.trim ( )`
If this object is not already a string, it is converted to a string. The result is a copy of the string with both leading and trailing white space removed. The definition of white space is the union of `WhiteSpace` and `Line Terminator`. The result is a string value, not a String object.

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

15.5.4.21 `String.prototype.toJSON (key)`
When the `toJSON` method is called with argument key, the following steps are taken:
1. Let O be this object.
2. Call the `[[Get]]` method of O with argument "valueOf".
3. If `IsCallable(Result(2))` is `false`, go to step 6.
4. Call the `[[Call]]` method of Result(2) with O as the this value and an empty argument list.
5. If Result(4) is a primitive value, return Result(4).
6. Throw a `TypeError` exception.

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

15.5.5 Properties of String Instances
String instances inherit properties from the String prototype object and also have a `[[PrimitiveValue]]` property and a `length` property.

The `[[PrimitiveValue]]` property is the string value represented by this String object.

15.5.5.1 `length`
The number of characters in the String value represented by this String object.

Once a String object is created, this property is unchanged. It has the attributes { `[[Writable]]`: `false`, `[[Enumerable]]`: `false`, `[[Configurable]]`: `false` }.

15.5.5.2 `[[GetOwnProperty]] (P)`
String objects use a variation of the `[[GetOwnProperty]]` method used for other native ECMAScript objects (8.6.2.8).
Assume $S$ is a String object and $P$ is a string.

When the `[[GetOwnProperty]]` method of $S$ is called with property name $P$, the following steps are taken:

1. Call the default `[[GetOwnProperty]]` method (8.6.2.8) with $S$ as the `this` value and argument $P$.
2. If Result(1) is not `undefined` return Result(1).
3. If $P$ is not an array index (15.4), return `undefined`.
4. Call `ToString`, giving $S$ as its argument.
5. Call `ToInteger(P)`.
6. Compute the number of characters in Result(4).
7. If Result(5) is less than 0 or is not less than Result(6), return `undefined`.
8. Create a string of length 1, containing one character from Result(4), namely the character at position Result(5), where the first (leftmost) character in Result(4) is considered to be at position 0, the next one at position 1, and so on.
9. Return a Property Descriptor `{ [[Value]]: Result(8), [[Enumerable]]: false, [[Writable]]: false, [[Configurable]]: false }`.

## 15.6 Boolean Objects

### 15.6.1 The Boolean Constructor Called as a Function

When `Boolean` is called as a function rather than as a constructor, it performs a type conversion.

#### 15.6.1.1 `Boolean (value)`

Returns a boolean value (not a Boolean object) computed by `ToBoolean(value)`.

### 15.6.2 The Boolean Constructor

When `Boolean` is called as part of a `new` expression it is a constructor: it initialise the newly created object.

#### 15.6.2.1 `new Boolean (value)`

The `[[Prototype]]` property of the newly constructed object is set to the original Boolean prototype object, the one that is the initial value of `Boolean.prototype` (15.6.3.1).

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

The `[[PrimitiveValue]]` property of the newly constructed Boolean object is set to `ToBoolean(value)`.

The `[[Extensible]]` property of the newly constructed object is set to `true`.

### 15.6.3 Properties of the Boolean Constructor

The value of the internal `[[Prototype]]` property of the Boolean constructor is the Function prototype object (15.3.4).

Besides the internal properties and the `length` property (whose value is 1), the Boolean constructor has the following property:

#### 15.6.3.1 `Boolean.prototype`

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

This property has the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

### 15.6.4 Properties of the Boolean Prototype Object

The Boolean prototype object is itself a Boolean object (its `[[Class]]` is "Boolean") whose value is `false`.

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

In following descriptions of functions that are properties of the Boolean prototype object, the phrase “this Boolean object” refers to the object that is the `this` value for the invocation of the function; a `TypeError` exception is thrown if the `this` value is not an object for which the value of the internal `[[Class]]` property is "Boolean". Also, the phrase “this boolean value” refers to the boolean value represented by this Boolean object, that is, the value of the internal `[[PrimitiveValue]]` property of this Boolean object.
15.6.4.1 Boolean.prototype.constructor
The initial value of Boolean.prototype.constructor is the built-in Boolean constructor.

15.6.4.2 Boolean.prototype.toString ( )
If this boolean value is true, then the string "true" is returned. Otherwise, this boolean value must be false, and the string "false" is returned.

The toString function is not generic; it throws a TypeError exception if its this value is not a Boolean object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.6.4.3 Boolean.prototype.valueOf ( )
Returns this boolean value.

The valueOf function is not generic; it throws a TypeError exception if its this value is not a Boolean object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.6.4.4 Boolean.prototype.toJSON ( key )
When the toJSON method is called with argument key, the following steps are taken:

1. Let O be this object.
2. Call the [[Get]] method of O with argument "valueOf".
3. If IsCallable(Result(2)) is false, go to step 6
4. Call the [[Call]] method of Result(2) with O as the this value and an empty argument list.
5. If Result(4) is a primitive value, return Result(4).
6. Throw a TypeError exception.

NOTE The toJSON function is intentionally generic; it does not require that its this value be a Boolean object. Therefore, it can be transferred to other kinds of objects for use as a method. An object is free to use the arguments ‘key’ that is passed in to filter its stringification.

15.6.5 Properties of Boolean Instances
Boolean instances have no special properties beyond those inherited from the Boolean prototype object.

15.7 Number Objects

15.7.1 The Number Constructor Called as a Function
When Number is called as a function rather than as a constructor, it performs a type conversion.

15.7.1.1 Number ( [ value ] )
Returns a number value (not a Number object) computed by ToNumber(value) if value was supplied, else returns +0.

15.7.2 The Number Constructor
When Number is called as part of a new expression it is a constructor; it initialises the newly created object.

15.7.2.1 new Number ( [ value ] )
The [[Prototype]] property of the newly constructed object is set to the original Number prototype object, the one that is the initial value of Number.prototype (15.7.3.1).

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

The [[PrimitiveValue]] property of the newly constructed object is set to ToNumber(value) if value was supplied, else to +0.

The [[Extensible]] property of the newly constructed object is set to true.

15.7.3 Properties of the Number Constructor
The value of the internal [[Prototype]] property of the Number constructor is the Function prototype object (15.3.4).

Besides the internal properties and the length property (whose value is 1), the Number constructor has the following property:
15.7.3.1 Number.prototype
The initial value of Number.prototype is the Number prototype object (15.7.4).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.7.3.2 Number.MAX_VALUE
The value of Number.MAX_VALUE is the largest positive finite value of the number type, which is approximately 1.7976931348623157 \times 10^{308}.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.7.3.3 Number.MIN_VALUE
The value of Number.MIN_VALUE is the smallest positive value of the number type, which is approximately 5 \times 10^{-324}.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.7.3.4 Number.NaN
The value of Number.NaN is NaN.

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.7.3.5 Number.NEGATIVE_INFINITY
The value of Number.NEGATIVE_INFINITY is \(-\infty\).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.7.3.6 Number.POSITIVE_INFINITY
The value of Number.POSITIVE_INFINITY is \(\infty\).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.7.4 Properties of the Number Prototype Object
The Number prototype object is itself a Number object (its [[Class]] is "Number") whose value is +0. The value of the internal [[Prototype]] property of the Number prototype object is the Object prototype object (15.2.3.1).

In following descriptions of functions that are properties of the Number prototype object, the phrase “this Number object” refers to the object that is the this value for the invocation of the function; a TypeError exception is thrown if the this value is not an object for which the value of the internal [[Class]] property is "Number". Also, the phrase “this number value” refers to the number value represented by this Number object, that is, the value of the internal [[PrimitiveValue]] property of this Number object.

15.7.4.1 Number.prototype.constructor
The initial value of Number.prototype.constructor is the built-in Number constructor.

15.7.4.2 Number.prototype.toString (radix)
If radix is the number 10 or undefined, then this number value is given as an argument to the ToString operator; the resulting string value is returned.

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

The toString function is not generic; it throws a TypeError exception if its this value is not a Number object. Therefore, it cannot be transferred to other kinds of objects for use as a method.

15.7.4.3 Number.prototype.toLocaleString()
Produces a string value that represents the value of the Number formatted according to the conventions of the host environment’s current locale. This function is implementation-dependent, and it is permissible, but not encouraged, for it to return the same thing as toString.

NOTE

15 December 2008
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

**15.7.4.4 Number.prototype.valueOf ( )**

Returns this number value.

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

**15.7.4.5 Number.prototype.toFixed (fractionDigits)**

Return a string containing the number represented in fixed-point notation with `fractionDigits` digits after the decimal point. If `fractionDigits` is `undefined`, 0 is assumed. Specifically, perform the following steps:

1. Let `f` be `ToInteger(fractionDigits)`. (If `fractionDigits` is `undefined`, this step produces the value 0).
2. If `f < 0` or `f > 20`, throw a `RangeError` exception.
3. Let `x` be this number value.
4. If `x` is `NaN`, return the string "NaN".
5. Let `s` be the empty string.
6. If `x ≤ 0`, go to step 9.
7. Let `s` be "-".
8. Let `x` = –`x`.
9. If `x ≥ 10^21`, let `m` = `ToString(x)` and go to step 20.
10. Let `n` be an integer for which the exact mathematical value of `n · 10^f` – `x` is as close to zero as possible. If there are two such `n`, pick the larger `n`.
11. If `n = 0`, let `m` be the string "0". Otherwise, let `m` be the string consisting of the digits of the decimal representation of `n` (in order, with no leading zeroes).
12. If `f = 0`, go to step 20.
13. Let `k` be the number of characters in `m`.
14. If `k > f`, go to step 18.
15. Let `z` be the string consisting of `f+1-k` occurrences of the character ‘0’.
16. Let `m` be the concatenation of strings `z` and `m`.
17. Let `k = f+1`.
18. Let `a` be the first `k-f` characters of `m`, and let `b` be the remaining `f` characters of `m`.
19. Let `m` be the concatenation of the three strings `a",",b`.
20. Return the concatenation of the strings `s` and `m`.

The `length` property of the `toFixed` method is 1.

If the `toFixed` method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of `toFixed` for values of `fractionDigits` less than 0 or greater than 20. In this case `toFixed` would not necessarily throw `RangeError` for such values.

**NOTE**

The output of `toFixed` may be more precise than `toString` for some values because `toString` only prints enough significant digits to distinguish the number from adjacent number values. For example, `(1000000000000000128).toString()` returns "1000000000000000100", while `(1000000000000000128).toFixed(0)` returns "1000000000000000128".

**15.7.4.6 Number.prototype.toExponential (fractionDigits)**

Return a string containing the number represented in exponential notation with one digit before the significand’s decimal point and `fractionDigits` digits after the significand’s decimal point. If `fractionDigits` is `undefined`, include as many significand digits as necessary to uniquely specify the number (just like in `toString` except that in this case the number is always output in exponential notation). Specifically, perform the following steps:

1. Let `x` be this number value.
2. Let `f` be `ToInteger(fractionDigits)`. 

15 December 2008
3. If \( x \) is `NaN`, return the string "`NaN`".
4. Let \( s \) be the empty string.
5. If \( x \leq 0 \), go to step 8.
6. Let \( s \) be "`-\n`".
7. Let \( x = -x \).
8. If \( x = +\infty \), let \( m = \text{"Infinity"} \) and go to step 30.
9. If `fractionDigits` is `undefined`, go to step 14.
10. If \( f < 0 \) or \( f > 20 \), throw a `RangeError` exception.
11. If \( x = 0 \), go to step 16.
12. Let \( e \) and \( n \) be integers such that \( 10^0 \leq n < 10^{+1} \) and for which the exact mathematical value of \( n \times 10^{-f} - x \) is as close to zero as possible. If there are two such sets of \( e \) and \( n \), pick the \( e \) and \( n \) for which \( n \times 10^{-f} \) is larger.
14. If \( x \neq 0 \), go to step 19.
15. Let \( f = 0 \).
16. Let \( m \) be the string consisting of \( f+1 \) occurrences of the character "`0`".
17. Let \( e = 0 \).
18. Go to step 21.
19. Let \( e \), \( n \), and \( f \) be integers such that \( f \geq 0 \), \( 10^0 \leq n < 10^{+1} \), the number value for \( n \times 10^{-f} \) is \( x \), and \( f \) is as small as possible. Note that the decimal representation of \( n \) has \( f+1 \) digits, \( n \) is not divisible by 10, and the least significant digit of \( n \) is not necessarily uniquely determined by these criteria.
20. Let \( m \) be the string consisting of the digits of the decimal representation of \( n \) (in order, with no leading zeroes).
21. If \( f = 0 \), go to step 24.
22. Let \( a \) be the first character of \( m \), and let \( b \) be the remaining \( f \) characters of \( m \).
23. Let \( m \) be the concatenation of the three strings \( a \), "`-`", and \( b \).
24. If \( e = 0 \), let \( c = "^+\" \) and \( d = "0\" \) and go to step 29.
25. If \( e > 0 \), let \( c = "+\" \) and go to step 28.
26. Let \( c = "-\" \).
27. Let \( e = e \).
28. Let \( d \) be the string consisting of the digits of the decimal representation of \( e \) (in order, with no leading zeroes).
29. Let \( m \) be the concatenation of the four strings \( a \), "`-`", \( c \), and \( d \).
30. Return the concatenation of the strings \( s \) and \( m \).

The `length` property of the `toExponential` method is 1.

If the `toExponential` method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of `toExponential` for values of `fractionDigits` less than 0 or greater than 20. In this case `toExponential` would not necessarily throw `RangeError` for such values.

**NOTE**
For implementations that provide more accurate conversions than required by the rules above, it is recommended that the following alternative version of step 19 be used as a guideline:

Let \( e \), \( n \), and \( f \) be integers such that \( f \geq 0 \), \( 10^0 \leq n < 10^{+1} \), the number value for \( n \times 10^{-f} \) is \( x \), and \( f \) is as small as possible. If there are multiple possibilities for \( n \), choose the value of \( n \) for which \( n \times 10^{-f} \) is closest in value to \( x \). If there are two such possible values of \( n \), choose the one that is even.

### 15.7.4.7 Number.prototype.toPrecision (precision)

Return a string containing the number represented either in exponential notation with one digit before the significand's decimal point and `precision`–1 digits after the significand's decimal point or in fixed notation with `precision` significant digits. If `precision` is `undefined`, call `toString` (9.8.1) instead.

Specifically, perform the following steps:

1. Let \( x \) be this number value.
2. If `precision` is `undefined`, return `toString(x)`.
3. Let \( p \) be `ToInteger(precision)`.
4. If \( x \) is NaN, return the string "NaN".
5. Let \( s \) be the empty string.
6. If \( x \geq 0 \), go to step 9.
7. Let \( s \) be " - ".
8. Let \( x = -x \).
9. If \( x = +\infty \), let \( m = "\text{Infinity}" \) and go to step 30.
10. If \( p < 1 \) or \( p > 21 \), throw a RangeError exception.
11. If \( x = 0 \), go to step 15.
12. Let \( m \) be the string consisting of \( p \) occurrences of the character ' 0 '.
13. Let \( e = 0 \).
15. Let \( e \) and \( n \) be integers such that \( 10^{p-1} \leq n < 10^p \) and for which the exact mathematical value of \( n \cdot 10^{p-e+1} - x \) is as close to zero as possible. If there are two such sets of \( e \) and \( n \), pick the \( e \) and \( n \) for which \( n \cdot 10^{e-p+1} \) is larger.
16. Let \( m \) be the string consisting of the digits of the decimal representation of \( n \) (in order, with no leading zeroes).
17. If \( e < -6 \) or \( e \geq p \), go to step 22.
18. If \( e = p-1 \), go to step 30.
19. If \( e \geq 0 \), let \( m \) be the concatenation of the first \( e+1 \) characters of \( m \), the character ' - ', and the remaining \( p-(e+1) \) characters of \( m \) and go to step 30.
20. Let \( m \) be the concatenation of the string " 0. \( ^{-} \),\( -(e+1) \) occurrences of the character ' 0 ' and the string \( m \).
22. Let \( a \) be the first character of \( m \), and let \( b \) be the remaining \( p-1 \) characters of \( m \).
23. Let \( m \) be the concatenation of the three strings \( a, " \cdot \), " \) and \( b \).
24. If \( e = 0 \), let \( c = "+" \) and \( d = "0" \) and go to step 29.
25. If \( e > 0 \), let \( c = "+" \) and go to step 28.
26. Let \( c = "-" \).
27. Let \( e = -e \).
28. Let \( d \) be the string consisting of the digits of the decimal representation of \( e \) (in order, with no leading zeroes).
29. Let \( m \) be the concatenation of the four strings \( m, "e", c, \) and \( d \).
30. Return the concatenation of the strings \( s \) and \( m \).

The length property of the toPrecision method is 1.

If the toPrecision method is called with more than one argument, then the behaviour is undefined (see clause 15).

An implementation is permitted to extend the behaviour of toPrecision for values of precision less than 1 or greater than 21. In this case toPrecision would not necessarily throw RangeError for such values.

15.7.4.8 Number.prototype.toJSON ( key )
When the toJSON method is called with argument key, the following steps are taken:
1. Let \( O \) be this object.
2. Call the [[Get]] method of \( O \) with argument "valueOf".
3. If IsCallable(Result(2)) is false, go to step 6
4. Call the [[Call]] method of Result(2) with \( O \) as the this value and an empty argument list.
5. If Result(4) is a primitive value, return Result(4).
6. Throw a TypeError exception.

NOTE
The toJSON function is intentionally generic; it does not require that its this value be a Number object. Therefore, it can be transferred to other kinds of objects for use as a method.
An object is free to use the argument 'key' that is passed in to filter its stringification.

15.7.5 Properties of Number Instances
Number instances have no special properties beyond those inherited from the Number prototype object.
15.8 The Math Object

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

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

The value of the internal [[Class]] property of the Math object is "Math".

The Math object does not have a [[Construct]] property; it is not possible to use the Math object as a constructor with the new operator.

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

NOTE
In this specification, the phrase “the number value for x” has a technical meaning defined in 8.5.

15.8.1 Value Properties of the Math Object

15.8.1.1 E
The number value for e, the base of the natural logarithms, which is approximately 2.7182818284590452354.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false, [[DontEnum]], [[DontDelete]], [[ReadOnly]] }.

15.8.1.2 LN10
The number value for the natural logarithm of 10, which is approximately 2.302585092994046.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false, [[DontEnum]], [[DontDelete]], [[ReadOnly]] }.

15.8.1.3 LN2
The number value for the natural logarithm of 2, which is approximately 0.6931471805599453.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false, [[DontEnum]], [[DontDelete]], [[ReadOnly]] }.

15.8.1.4 LOG2E
The number value for the base-2 logarithm of e, the base of the natural logarithms; this value is approximately 1.4426950408889634.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false, [[DontEnum]], [[DontDelete]], [[ReadOnly]] }.

NOTE
The value of Math.LOG2E is approximately the reciprocal of the value of Math.LN2.

15.8.1.5 LOG10E
The number value for the base-10 logarithm of e, the base of the natural logarithms; this value is approximately 0.4342944819032518.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false, [[DontEnum]], [[DontDelete]], [[ReadOnly]] }.

NOTE
The value of Math.LOG10E is approximately the reciprocal of the value of Math.LN10.

15.8.1.6 PI
The number value for π, the ratio of the circumference of a circle to its diameter, which is approximately 3.1415926535897932.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false, [[DontEnum]], [[DontDelete]], [[ReadOnly]] }.

15.8.1.7 SQRT1_2
The number value for the square root of 1/2, which is approximately 0.7071067811865476.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false, [[DontEnum]], [[DontDelete]], [[ReadOnly]] }.

NOTE
The value of Math.SQRT1_2 is approximately the reciprocal of the value of Math.SQRT2.

15.8.1.8 SQRT2
The number value for the square root of 2, which is approximately 1.4142135623730951.
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false, [[DontEnum]], [[DontDelete]], [[ReadOnly]] }.
15.8.2 Function Properties of the Math Object
Every function listed in this section applies the ToNumber operator to each of its arguments (in left-to-right order if there is more than one) and then performs a computation on the resulting number value(s).

In the function descriptions below, the symbols NaN, -0, +0, -\infty and +\infty refer to the number values described in 8.5.

**NOTE**
The behaviour of the functions acos, asin, atan, atan2, cos, exp, log, pow, sin, and sqrt is not precisely specified here except to require specific results for certain argument values that represent boundary cases of interest. For other argument values, these functions are intended to compute approximations to the results of familiar mathematical functions, but some latitude is allowed in the choice of approximation algorithms. The general intent is that an implementer should be able to use the same mathematical library for ECMAScript on a given hardware platform that is available to C programmers on that platform.

Although the choice of algorithms is left to the implementation, it is recommended (but not specified by this standard) that implementations use the approximation algorithms for IEEE 754 arithmetic contained in *fdlibm*, the freely distributable mathematical library from Sun Microsystems ([fdlibm-comment@sunpro.eng.sun.com](mailto:fdlibm-comment@sunpro.eng.sun.com)). This specification also requires specific results for certain argument values that represent boundary cases of interest.

15.8.2.1 abs (x)
Returns the absolute value of x; the result has the same magnitude as x but has positive sign.
If x is NaN, the result is NaN.
If x is -0, the result is +0.
If x is -\infty, the result is +\infty.

15.8.2.2 acos (x)
Returns an implementation-dependent approximation to the arc cosine of x. The result is expressed in radians and ranges from +0 to +\pi.
If x is NaN, the result is NaN.
If x is greater than 1, the result is NaN.
If x is less than -1, the result is NaN.
If x is +0, the result is +0.

15.8.2.3 asin (x)
Returns an implementation-dependent approximation to the arc sine of x. The result is expressed in radians and ranges from -\pi/2 to +\pi/2.
If x is NaN, the result is NaN.
If x is greater than 1, the result is NaN.
If x is less than -1, the result is NaN.
If x is +0, the result is +0.
If x is -0, the result is -0.

15.8.2.4 atan (x)
Returns an implementation-dependent approximation to the arc tangent of x. The result is expressed in radians and ranges from -\pi/2 to +\pi/2.
If x is NaN, the result is NaN.
If x is +0, the result is +0.
If x is -0, the result is -0.
If x is +\infty, the result is an implementation-dependent approximation to +\pi/2.
If x is -\infty, the result is an implementation-dependent approximation to -\pi/2.
15.8.2.5 atan2 (y, x)

Returns an implementation-dependent approximation to the arc tangent of the quotient y/x of the arguments y and x, where the signs of y and x are used to determine the quadrant of the result. Note that it is intentional and traditional for the two-argument arc tangent function that the argument named y be first and the argument named x be second. The result is expressed in radians and ranges from $-\pi$ to $+\pi$.

If either x or y is NaN, the result is NaN.
If y > 0 and x is +0, the result is an implementation-dependent approximation to $+\pi/2$.
If y > 0 and x is -0, the result is an implementation-dependent approximation to $+\pi/2$.
If y is +0 and x > 0, the result is +0.
If y is +0 and x is +0, the result is +0.
If y is +0 and x is -0, the result is an implementation-dependent approximation to $+\pi$.
If y is +0 and x is 0, the result is an implementation-dependent approximation to $+\pi$.
If y is -0 and x > 0, the result is an implementation-dependent approximation to $-\pi$.
If y is -0 and x is +0, the result is +0.
If y is -0 and x is -0, the result is an implementation-dependent approximation to $-\pi$.
If y is +0 and x < 0, the result is an implementation-dependent approximation to $-\pi$.
If y is +0 and x is 0, the result is an implementation-dependent approximation to $-\pi$.
If y < 0 and x is +0, the result is an implementation-dependent approximation to $-\pi/2$.
If y < 0 and x is 0, the result is an implementation-dependent approximation to $-\pi/2$.
If x > 0 and y is finite and x is $+\infty$, the result is +0.
If x > 0 and y is finite and x is $-\infty$, the result is 1.
If y < 0 and y is finite and x is $-\infty$, the result is an implementation-dependent approximation to $+\pi$.
If y < 0 and y is finite and x is $+\infty$, the result is an implementation-dependent approximation to $+\pi$.
If y < 0 and y is finite and x is $+\infty$, the result is an implementation-dependent approximation to $+\pi/2$.
If y < 0 and y is finite and x is $-\infty$, the result is an implementation-dependent approximation to $+\pi/2$.
If y < 0 and y is finite and x is $-\infty$, the result is an implementation-dependent approximation to $+\pi/4$.
If y < 0 and y is finite and x is $+\infty$, the result is an implementation-dependent approximation to $+3\pi/4$.
If y < 0 and y is finite and x is $-\infty$, the result is an implementation-dependent approximation to $+3\pi/4$.

15.8.2.6 ceil (x)

Returns the smallest (closest to $+\infty$) number value that is not less than x and is equal to a mathematical integer. If x is already an integer, the result is x.

If x is NaN, the result is NaN.
If x is +0, the result is +0.
If x is -0, the result is -0.
If x is $+\infty$, the result is $+\infty$.
If x is $-\infty$, the result is $-\infty$.
If x is less than 0 but greater than $-1$, the result is $-0$.

The value of Math.ceil(x) is the same as the value of -Math.floor(-x).

15.8.2.7 cos (x)

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

If x is NaN, the result is NaN.
If x is +0, the result is 1.
If x is -0, the result is 1.
If x is $+\infty$, the result is NaN.

15 December 2008
If $x$ is $-\infty$, the result is NaN.

15.8.2.8 **exp (x)**

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

If $x$ is NaN, the result is NaN.
If $x$ is $+0$, the result is $1$.
If $x$ is $-0$, the result is $1$.
If $x$ is $+\infty$, the result is $+\infty$.
If $x$ is $-\infty$, the result is $+0$.

15.8.2.9 **floor (x)**

Returns the greatest (closest to $+\infty$) number value that is not greater than $x$ and is equal to a mathematical integer. If $x$ is already an integer, the result is $x$.

If $x$ is NaN, the result is NaN.
If $x$ is $+0$, the result is $+0$.
If $x$ is $-0$, the result is $-0$.
If $x$ is $+\infty$, the result is $+\infty$.
If $x$ is $-\infty$, the result is $-\infty$.
If $x$ is greater than $0$ but less than $1$, the result is $+0$.

*NOTE*

The value of Math.floor(x) is the same as the value of -Math.ceil(-x).

15.8.2.10 **log (x)**

Returns an implementation-dependent approximation to the natural logarithm of $x$.

If $x$ is NaN, the result is NaN.
If $x$ is less than $0$, the result is NaN.
If $x$ is $+0$ or $0$, the result is $0$.
If $x$ is $1$, the result is $+0$.
If $x$ is $+\infty$, the result is $+\infty$.

15.8.2.11 **max ( [ value1 [ , value2 [ , ... ] ] ] )**

Given zero or more arguments, calls ToNumber on each of the arguments and returns the largest of the resulting values.

If no arguments are given, the result is $-\infty$.
If any value is NaN, the result is NaN.

The comparison of values to determine the largest value is done as in 11.8.5 except that $+0$ is considered to be larger than $-0$.

The `length` property of the `max` method is $2$.

15.8.2.12 **min ( [ value1 [ , value2 [ , ... ] ] ] )**

Given zero or more arguments, calls ToNumber on each of the arguments and returns the smallest of the resulting values.

If no arguments are given, the result is $+\infty$.
If any value is NaN, the result is NaN.

The comparison of values to determine the smallest value is done as in 11.8.5 except that $+0$ is considered to be larger than $-0$.

The `length` property of the `min` method is $2$.
15.8.2.13 pow (x, y)
Returns an implementation-dependent approximation to the result of raising x to the power y.
If y is NaN, the result is NaN.
If y is +0, the result is 1, even if x is NaN.
If y is –0, the result is 1, even if x is NaN.
If x is NaN and y is nonzero, the result is NaN.
If abs(x)>1 and y is +0, the result is +∞.
If abs(x)>1 and y is –0, the result is +0.
If abs(x)=1 and y is +0, the result is NaN.
If abs(x)=1 and y is –0, the result is NaN.
If abs(x)<1 and y is +0, the result is +0.
If abs(x)<1 and y is –0, the result is +0.
If x is +0 and y>0, the result is +0.
If x is +0 and y<0, the result is +0.
If x is –0 and y>0 and y is an odd integer, the result is –0.
If x is –0 and y>0 and y is not an odd integer, the result is +0.
If x is –0 and y<0 and y is an odd integer, the result is –0.
If x is –0 and y<0 and y is not an odd integer, the result is +0.
If x is +0 and y>0, the result is +0.
If x is +0 and y<0, the result is +0.
If x<0 and x is finite and y is not an integer, the result is NaN.

15.8.2.14 random ( )
Returns a number value with positive sign, greater than or equal to 0 but less than 1, chosen randomly or pseudo randomly with approximately uniform distribution over that range, using an implementation-dependent algorithm or strategy. This function takes no arguments.

15.8.2.15 round (x)
Returns the number value that is closest to x and is equal to a mathematical integer. If two integer number values are equally close to x, then the result is the number value that is closer to +0. If x is already an integer, the result is x.
If x is NaN, the result is NaN.
If x is +0, the result is +0.
If x is –0, the result is –0.
If x is +0, the result is +0.
If x is –0, the result is –0.
If x is greater than 0 but less than 0.5, the result is +0.
If x is less than 0 but greater than or equal to –0.5, the result is –0.

NOTE 1
Math.round(3.5) returns 4, but Math.round(–3.5) returns –3.
NOTE 2
The value of Math.round(x) is the same as the value of Math.floor(x+0.5), except when x is 0 or is less than 0 but greater than or equal to –0.5; for these cases Math.round(x) returns 0, but Math.floor(x+0.5) returns +0.
15.8.2.16 **sin (x)**

Returns an implementation-dependent approximation to the sine of \( x \). The argument is expressed in radians.

- If \( x \) is NaN, the result is NaN.
- If \( x \) is +0, the result is +0.
- If \( x \) is –0, the result is –0.
- If \( x \) is +\( \infty \) or –\( \infty \), the result is NaN.

15.8.2.17 **sqrt (x)**

Returns an implementation-dependent approximation to the square root of \( x \).

- If \( x \) is NaN, the result is NaN.
- If \( x \) less than 0, the result is NaN.
- If \( x \) is +0, the result is +0.
- If \( x \) is –0, the result is –0.
- If \( x \) is +\( \infty \), the result is +\( \infty \).

15.8.2.18 **tan (x)**

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

- If \( x \) is NaN, the result is NaN.
- If \( x \) is +0, the result is +0.
- If \( x \) is –0, the result is –0.
- If \( x \) is +\( \infty \) or –\( \infty \), the result is NaN.

15.9 **Date Objects**

15.9.1 **Overview of Date Objects and Definitions of Internal Operators**

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

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

15.9.1.1 **Time Range**

Time is measured in ECMAScript in milliseconds since 01 January, 1970 UTC. Leap seconds are ignored. It is assumed that there are exactly 86,400,000 milliseconds per day. ECMAScript number values can represent all integers from \(-9,007,199,254,740,991\) to \(9,007,199,254,740,991\); this range suffices to measure times to millisecond precision for any instant that is within approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC.

The actual range of times supported by ECMAScript Date objects is slightly smaller: exactly \(-100,000,000\) days to \(100,000,000\) days measured relative to midnight at the beginning of 01 January, 1970 UTC. This gives a range of \(8,640,000,000,000,000\) milliseconds to either side of 01 January, 1970 UTC.

The exact moment of midnight at the beginning of 01 January, 1970 UTC is represented by the value +0.

15.9.1.2 **Day Number and Time within Day**

A given time value \( t \) belongs to day number

\[
\text{Day}(t) = \text{floor}(\frac{t}{\text{msPerDay}})
\]

where the number of milliseconds per day is

\[
\text{msPerDay} = 86400000
\]

The remainder is called the time within the day:

15 December 2008
TimeWithinDay(t) = t modulo msPerDay

15.9.1.3 Year Number

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

\[
\text{DaysInYear}(y) = \begin{cases} 
365 & \text{if } (y \mod 4) \neq 0 \\
366 & \text{if } (y \mod 4) = 0 \text{ and } (y \mod 100) \neq 0 \\
365 & \text{if } (y \mod 100) = 0 \text{ and } (y \mod 400) \neq 0 \\
366 & \text{if } (y \mod 400) = 0
\end{cases}
\]

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

\[
\text{DayFromYear}(y) = 365 \times (y - 1970) + \lfloor(y - 1969)/4\rfloor - \lfloor(y - 1901)/100\rfloor + \lfloor(y - 1601)/400\rfloor
\]

The time value of the start of a year is:

\[
\text{TimeFromYear}(y) = \text{msPerDay} \times \text{DayFromYear}(y)
\]

15.9.1.4 Month Number

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

\[
\text{MonthFromTime}(t) = \begin{cases} 
0 & \text{if DayWithinYear}(t) < 31 \\
1 & \text{if DayWithinYear}(t) < 59 + \text{InLeapYear}(t) \\
2 & \text{if DayWithinYear}(t) < 90 + \text{InLeapYear}(t) \\
3 & \text{if DayWithinYear}(t) < 120 + \text{InLeapYear}(t) \\
4 & \text{if DayWithinYear}(t) < 151 + \text{InLeapYear}(t) \\
5 & \text{if DayWithinYear}(t) < 181 + \text{InLeapYear}(t) \\
6 & \text{if DayWithinYear}(t) < 212 + \text{InLeapYear}(t) \\
7 & \text{if DayWithinYear}(t) < 243 + \text{InLeapYear}(t) \\
8 & \text{if DayWithinYear}(t) < 273 + \text{InLeapYear}(t) \\
9 & \text{if DayWithinYear}(t) < 304 + \text{InLeapYear}(t) \\
10 & \text{if DayWithinYear}(t) < 334 + \text{InLeapYear}(t) \\
11 & \text{if DayWithinYear}(t) < 365 + \text{InLeapYear}(t)
\end{cases}
\]

where

\[
\text{DayWithinYear}(t) = \text{Day}(t) - \text{DayFromYear}(\text{YearFromTime}(t))
\]

A month value of 0 specifies January; 1 specifies February; 2 specifies March; 3 specifies April; 4 specifies May; 5 specifies June; 6 specifies July; 7 specifies August; 8 specifies September; 9 specifies October; 10 specifies November; and 11 specifies December. Note that MonthFromTime(0) = 0, corresponding to Thursday, 01 January, 1970.

15.9.1.5 Date Number

A date number is identified by an integer in the range 1 through 31, inclusive. The mapping DateFromTime(t) from a time value t to a month number is defined by:

\[
\text{DateFromTime}(t) = \begin{cases} 
\text{DayWithinYear}(t) + 1 & \text{if MonthFromTime}(t) = 0 \\
\text{DayWithinYear}(t) & \text{if MonthFromTime}(t) = 1 \\
\text{DayWithinYear}(t) - 30 & \text{if MonthFromTime}(t) = 2
\end{cases}
\]

15 December 2008
15.9.1.6 Week Day

The weekday for a particular time value \( t \) is defined as

\[
\text{WeekDay}(t) = (\text{Day}(t) + 4) \mod 7
\]

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

15.9.1.7 Local Time Zone Adjustment

An implementation of ECMAScript is expected to determine the local time zone adjustment. The local time zone adjustment is a value LocalTZA measured in milliseconds which when added to UTC represents the local standard time. Daylight saving time is not reflected by LocalTZA. The value LocalTZA does not vary with time but depends only on the geographic location.

15.9.1.8 Daylight Saving Time Adjustment

An implementation of ECMAScript is expected to determine the daylight saving time algorithm. The algorithm to determine the daylight saving time adjustment DaylightSavingTA(\( t \)), measured in milliseconds, must depend only on four things:

1. the time since the beginning of the year \( t - \text{TimeFromYear}(	ext{YearFromTime}(t)) \)
2. whether \( t \) is in a leap year \( \text{InLeapYear}(t) \)
3. the week day of the beginning of the year \( \text{WeekDay}(	ext{TimeFromYear}(	ext{YearFromTime}(t))) \) and
4. the geographic location.

The implementation of ECMAScript should not try to determine whether the exact time was subject to daylight saving time, but just whether daylight saving time would have been in effect if the current daylight saving time algorithm had been used at the time. This avoids complications such as taking into account the years that the locale observed daylight saving time year round.

If the host environment provides functionality for determining daylight saving time, the implementation of ECMAScript is free to map the year in question to an equivalent year (same leap-year-ness and same starting week day for the year) for which the host environment provides daylight saving time information. The only restriction is that all equivalent years should produce the same result.

15.9.1.9 Local Time

Conversion from UTC to local time is defined by

\[
\text{LocalTime}(t) = t + \text{LocalTZA} + \text{DaylightSavingTA}(t)
\]

Conversion from local time to UTC is defined by

\[
\text{UTC}(t) = t - \text{LocalTZA} - \text{DaylightSavingTA}(t - \text{LocalTZA})
\]

Note that UTC(LocalTime(\( t \))) is not necessarily always equal to \( t \).
15.9.1.10 Hours, Minutes, Second, and Milliseconds

The following functions are useful in decomposing time values:

\[
\text{HourFromTime}(t) = \text{floor}(t / \text{msPerHour}) \mod \text{HoursPerDay}
\]

\[
\text{MinFromTime}(t) = \text{floor}(t / \text{msPerMinute}) \mod \text{MinutesPerHour}
\]

\[
\text{SecFromTime}(t) = \text{floor}(t / \text{msPerSecond}) \mod \text{SecondsPerMinute}
\]

\[
\text{msFromTime}(t) = t \mod \text{msPerSecond}
\]

where

\[
\text{HoursPerDay} = 24
\]

\[
\text{MinutesPerHour} = 60
\]

\[
\text{SecondsPerMinute} = 60
\]

\[
\text{msPerSecond} = 1000
\]

\[
\text{msPerMinute} = \text{msPerSecond} \times \text{SecondsPerMinute} = 60000
\]

\[
\text{msPerHour} = \text{msPerMinute} \times \text{MinutesPerHour} = 3600000
\]

15.9.1.11 MakeTime (hour, min, sec, ms)

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

1. If \( \text{hour} \) is not finite or \( \text{min} \) is not finite or \( \text{sec} \) is not finite or \( \text{ms} \) is not finite, return NaN.
2. Call ToInteger(\( \text{hour} \)).
3. Call ToInteger(\( \text{min} \)).
4. Call ToInteger(\( \text{sec} \)).
5. Call ToInteger(\( \text{ms} \)).
6. Compute Result(2) \* \( \text{msPerHour} \) + Result(3) \* \( \text{msPerMinute} \) + Result(4) \* \( \text{msPerSecond} \) + Result(5), performing the arithmetic according to IEEE 754 rules (that is, as if using the ECMAScript operators \( \* \) and \( + \)).
7. Return Result(6).

15.9.1.12 MakeDay (year, month, date)

The operator MakeDay calculates a number of days from its three arguments, which must be ECMAScript number values. This operator functions as follows:

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

15.9.1.13 MakeDate (day, time)

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

1. If \( \text{day} \) is not finite or \( \text{time} \) is not finite, return NaN.
2. Compute \( \text{day} \times \text{msPerDay} + \text{time} \).
3. Return Result(2).
15.9.1.14 **TimeClip (time)**

The operator `TimeClip` calculates a number of milliseconds from its argument, which must be an ECMAScript number value. This operator functions as follows:

1. If `time` is not finite, return `NaN`.
2. If `abs(Result(1)) > 8.64 x 10^{15}`, return `NaN`.
3. Return an implementation-dependent choice of either `ToInteger(Result(2))` or `ToInteger(Result(2)) + (+0)`.

(Adding a positive zero converts `-0` to `+0`.)

**NOTE**

The point of step 3 is that an implementation is permitted a choice of internal representations of `time` values, for example as a 64-bit signed integer or as a 64-bit floating-point value. Depending on the implementation, this internal representation may or may not distinguish `-0` and `+0`.

15.9.1.15 **Date Time string format**

The Simplified ISO 8601 format is as follows: `YYYY-MM-DDTHH:mm:ss.sssTZ`

Where the fields are as follows:

- `YYYY` is the year in the Gregorian calendar
- `MM` is the month of the year between 01 (January) and 12 (December)
- `DD` is the day of the month between 01 and 31.
- The “T” appears literally in the string, to indicate the beginning of the time element, as specified in ISO 8601.
- `HH` is the number of complete hours that have passed since midnight
- `mm` is the number of complete minutes since the start of the hour
- `ss` is the number of complete seconds since the start of the minute
- The `.` (dot) `sss` is the number of complete milliseconds since the start of the second.
- Both the `.` and the milliseconds field are optional
- `TZ` is the timezone specified as `Z` (for UTC) or either `+` or `−` followed by a time expression `HH:MM`

**Extended years**

ECMAScript requires the ability to specify 6 digit years (extended years); approximately 285,616 years, either forward or backward, from 01 January, 1970 UTC. To represent years before 0 or after 9999, ISO 8601 permits the expansion of the year representation, but only by prior agreement between the sender and the receiver. In this simplified ECMAScript format such an expanded year representation shall have 2 extra year digits and is always prefixed with a `+` or `−` sign with the convention that year 0 is positive.

**NOTE**

This format includes date-only forms:

- `YYYY`
- `YYYY-MM`
- `YYYY-MM-DD`

Time-only forms with an optional time zone appended:

- `THH:mm`
- `THH:mm:ss`
- `THH:mm:ss.sss`

It also includes "date-times" which could be any combination of the above.

All numbers must be base 10.
Illegal values (out-of-bounds as well as syntax errors) in a format string means that the format string is not a valid instance of this format.

As every day both starts and ends with midnight, the two notations 00:00 and 24:00 are available to distinguish the two midnights that can be associated with one date. This means that the following two notations refer to exactly the same point in time: 1995-02-04T24:00 and 1995-02-05T00:00.

There exists no international standard that specifies abbreviations for civil time zones like CET, EST, etc. and sometimes the same abbreviation is even used for two very different time zones. For this reason, ISO 8601 and this format specifies numeric representations of date and time.

15.9.2 The Date Constructor Called as a Function
When `Date` is called as a function rather than as a constructor, it returns a string representing the current time (UTC).

**NOTE**
The function call `Date(...)` is not equivalent to the object creation expression `new Date(...)` with the same arguments.

15.9.2.1 `Date([year[, month[, date[, hours[, minutes[, seconds[, ms]]]]]])`
All of the arguments are optional; any arguments supplied are accepted but are completely ignored. A string is created and returned as if by the expression `new Date().toString()` where `Date` is the standard built-in constructor with that name and `toString` is the standard built-in method `Date.prototype.toString`.

15.9.3 The Date Constructor
When `Date` is called as part of a `new` expression, it is a constructor: it initialises the newly created object.

15.9.3.1 `new Date(year, month[, date[, hours[, minutes[, seconds[, ms]]]]])`
When `Date` is called with two to seven arguments, it computes the date from `year`, `month`, and (optionally) `date`, `hours`, `minutes`, `seconds` and `ms`.

The `[[Prototype]]` property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of `Date.prototype` (15.9.4.1).

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

The `[[Extensible]]` property of the newly constructed object is set to `true`.

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

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

15.9.3.2 `new Date(value)`
The `[[Prototype]]` property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of `Date.prototype` (15.9.4.1).

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

15 December 2008
The [[Extensible]] property of the newly constructed object is set to true.

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

1. Call ToPrimitive(value).
2. If Type(Result(1)) is String, then go to step 5.
3. Let V be ToNumber(Result(1)).
4. Set the [[PrimitiveValue]] property of the newly constructed object to TimeClip(V) and return.
5. Parse Result(1) as a date, in exactly the same manner as for the parse method (15.9.4.2); let V be the time value for this date.

15.9.3.3 new Date()

The [[Prototype]] property of the newly constructed object is set to the original Date prototype object, the one that is the initial value of Date.prototype (15.9.4.1).

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

The [[Extensible]] property of the newly constructed object is set to true.

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

15.9 Properties of the Date Constructor

The value of the internal [[Prototype]] property of the Date constructor is the Function prototype object (15.3.4).

Besides the internal properties and the length property (whose value is 7), the Date constructor has the following properties:

15.9.4.1 Date.prototype

The initial value of Date.prototype is the built-in Date prototype object (15.9.5).

This property has the attributes {{Writable]: false, [Enumerable]: false, [Configurable]: false}.

15.9.4.2 Date.parse (string)

The parse function applies the ToString operator to its argument and interprets the resulting string as a date; it returns a number, the UTC time value corresponding to the date. The string may be interpreted as a local time, a UTC time, or a time in some other time zone, depending on the contents of the string. The function first attempts to parse the format of the string according to the rules called out in Date Time String Format (15.9.1.15). If the string does not conform to that format, the function may fall back to any implementation-specific heuristics or implementation-specific date formats. Unrecognizable strings or dates containing illegal element values in the format string shall cause Date.parse to return NaN.

If x is any Date object whose milliseconds amount is zero within a particular implementation of ECMAScript, then all of the following expressions should produce the same numeric value in that implementation, if all the properties referenced have their initial values:

\[
\begin{align*}
x\text{.valueOf}() \\
\text{Date.parse}(x\text{.toString}()) \\
\text{Date.parse}(x\text{.toUTCString}())
\end{align*}
\]

However, the expression

\[
\text{Date.parse}(x\text{.toLocaleString}())
\]

is not required to produce the same number value as the preceding three expressions and, in general, the value produced by Date.parse is implementation-dependent when given any string value that could not be produced in that implementation by the toString or toUTCString method.
15.9.4.3 Date.UTC (year, month [, date [, hours [, minutes [, seconds [, ms ] ] ] ] ] )

When the UTC function is called with fewer than two arguments, the behaviour is implementation-dependent. When the UTC function is called with two to seven arguments, it computes the date from year, month and (optionally) date, hours, minutes, seconds and ms. The following steps are taken:

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

The length property of the UTC function is 7.

NOTE
The UTC function differs from the Date constructor in two ways: it returns a time value as a number, rather than creating a Date object, and it interprets the arguments in UTC rather than as local time.

15.9.4.4 Date.now ( )

The now method produces the time value at the time of the call.

15.9.5 Properties of the Date Prototype Object

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

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

In following descriptions of functions that are properties of the Date prototype object, the phrase “this Date object” refers to the object that is the this value for the invocation of the function. None of these functions are generic; a TypeError exception is thrown if the this value is not an object for which the value of the internal [[Class]] property is "Date". Also, the phrase “this time value” refers to the number value for the time represented by this Date object, that is, the value of the internal [[PrimitiveValue]] property of this Date object.

15.9.5.1 Date.prototype.constructor

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

15.9.5.2 Date.prototype.toString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form.

NOTE: For any Date value d whose milliseconds amount is zero, the result of Date.parse(d.toString()) is equal to d.valueOf(). See section 15.9.4.2.

15.9.5.3 Date.prototype.toDateString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the “date” portion of the Date in the current time zone in a convenient, human-readable form.

15.9.5.4 Date.prototype.toTimeString ( )

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the “time” portion of the Date in the current time zone in a convenient, human-readable form.
15.9.5.5  **Date.prototype.toLocaleString ( )**  
This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment’s current locale.

*NOTE*  
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.5.6  **Date.prototype.toLocaleDateString ( )**  
This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the “date” portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment’s current locale.

*NOTE*  
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.5.7  **Date.prototype.toLocaleTimeString ( )**  
This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the “time” portion of the Date in the current time zone in a convenient, human-readable form that corresponds to the conventions of the host environment’s current locale.

*NOTE*  
The first parameter to this function is likely to be used in a future version of this standard; it is recommended that implementations do not use this parameter position for anything else.

15.9.5.8  **Date.prototype.valueOf ( )**  
The `valueOf` function returns a number, which is this time value.

15.9.5.9  **Date.prototype.getTime ( )**  
1. If the `this` value is not an object whose `[[Class]]` property is "Date", throw a *TypeError* exception.
2. Return this time value.

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

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

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

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

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

15 December 2008
15.9.5.15 Date.prototype.getUTCDate ()
   1. Let t be this time value.
   2. If t is NaN, return NaN.
   3. Return DateFromTime(t).

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

15.9.5.17 Date.prototype.getUTCDay ()
   1. Let t be this time value.
   2. If t is NaN, return NaN.
   3. Return WeekDay(t).

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

15.9.5.19 Date.prototype.getUTCHours ()
   1. Let t be this time value.
   2. If t is NaN, return NaN.
   3. Return HourFromTime(t).

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

15.9.5.21 Date.prototype.getUTCMinutes ()
   1. Let t be this time value.
   2. If t is NaN, return NaN.
   3. Return MinFromTime(t).

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

15.9.5.23 Date.prototype.getUTCSecs ()
   1. Let t be this time value.
   2. If t is NaN, return NaN.
   3. Return SecFromTime(t).

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

15.9.5.25 Date.prototype.getUTCMilliseconds ()
   1. Let t be this time value.
   2. If t is NaN, return NaN.
   3. Return msFromTime(t).
15.9.5.26 Date.prototype.getTimezoneOffset ( )
Returns the difference between local time and UTC time in minutes.
1. Let t be this time value.
2. If t is NaN, return NaN.
3. Return (t - LocalTime(t)) / msPerMinute.

15.9.5.27 Date.prototype.setTime (time)
1. If the this value is not a Date object, throw a TypeError exception.
2. Call ToNumber(time).
3. Call TimeClip(Result(1)).
4. Set the [[PrimitiveValue]] property of the this value to Result(2).
5. Return the value of the [[PrimitiveValue]] property of the this value.

15.9.5.28 Date.prototype.setMilliseconds (ms)
1. Let t be the result of LocalTime(this time value).
2. Call ToNumber(ms).
3. Compute MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), Result(2)).
4. Compute UTC(MakeDate(Day(t), Result(3))).
5. Set the [[PrimitiveValue]] property of the this value to TimeClip(Result(4)).
6. Return the value of the [[PrimitiveValue]] property of the this value.

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

15.9.5.30 Date.prototype.setSeconds (sec [, ms ] )
If ms is not specified, this behaves as if ms were specified with the value getMilliseconds( ).
1. Let t be the result of LocalTime(this time value).
2. Call ToNumber(sec).
3. If ms is not specified, compute msFromTime(t); otherwise, call ToNumber(ms).
4. Compute MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), Result(2), Result(3)).
5. Compute UTC(MakeDate(Day(t), Result(4))).
6. Set the [[PrimitiveValue]] property of the this value to TimeClip(Result(5)).
7. Return the value of the [[PrimitiveValue]] property of the this value.

The length property of the setSeconds method is 2.

15.9.5.31 Date.prototype.setUTCSeconds (sec [, ms ] )
If ms is not specified, this behaves as if ms were specified with the value getUTCMilliseconds( ).
1. Let t be this time value.
2. Call ToNumber(sec).
3. If ms is not specified, compute msFromTime(t); otherwise, call ToNumber(ms).
4. Compute MakeTime(HourFromTime(t), MinFromTime(t), SecFromTime(t), Result(2), Result(3)).
5. Compute MakeDate(Day(t), Result(4)).
6. Set the [[PrimitiveValue]] property of the this value to TimeClip(Result(5)).
7. Return the value of the [[PrimitiveValue]] property of the this value.

The length property of the setUTCSeconds method is 2.

15.9.5.33 Date.prototype.setMinutes (min [, sec [, ms ] ] )
If sec is not specified, this behaves as if sec were specified with the value getSeconds( ).
If ms is not specified, this behaves as if ms were specified with the value getMilliseconds( ).
1. Let \( t \) be the result of LocalTime(this time value).
2. Call ToNumber(min).
3. If sec is not specified, compute SecFromTime(t); otherwise, call ToNumber(sec).
4. If ms is not specified, compute msFromTime(t); otherwise, call ToNumber(ms).
5. Compute MakeTime(HourFromTime(t), Result(2), Result(3), Result(4)).
6. Compute UTC(MakeDate(Day(t), Result(5))).
7. Set the [[PrimitiveValue]] property of the this value to TimeClip(Result(6)).
8. Return the value of the [[PrimitiveValue]] property of the this value.

1. Let \( t \) be the result of LocalTime(this time value).
2. Call ToNumber(hour).
3. If min is not specified, compute MinFromTime(t); otherwise, call ToNumber(min).
4. If sec is not specified, compute SecFromTime(t); otherwise, call ToNumber(sec).
5. If ms is not specified, compute msFromTime(t); otherwise, call ToNumber(ms).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute UTC(MakeDate(Day(t), Result(6))).
8. Set the [[PrimitiveValue]] property of the this value to TimeClip(Result(7)).
9. Return the value of the [[PrimitiveValue]] property of the this value.

The `length` property of the `setMinutes` method is 3.

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

The `length` property of the `setUTCHours` method is 4.

1. Let \( t \) be the result of LocalTime(this time value).
2. Call ToNumber(hour).
3. If min is not specified, compute MinFromTime(t); otherwise, call ToNumber(min).
4. If sec is not specified, compute SecFromTime(t); otherwise, call ToNumber(sec).
5. If ms is not specified, compute msFromTime(t); otherwise, call ToNumber(ms).
6. Compute MakeTime(Result(2), Result(3), Result(4), Result(5)).
7. Compute UTC(MakeDate(Day(t), Result(6))).
8. Set the [[PrimitiveValue]] property of the this value to TimeClip(Result(7)).
9. Return the value of the [[PrimitiveValue]] property of the this value.
The `length` property of the `setUTCHours` method is 4.

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

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

15.9.5.38 `Date.prototype.setMonth (month [, date ] )`
If `date` is not specified, this behaves as if `date` were specified with the value `getDate()`.
1. Let `t` be the result of `LocalTime(this time value)`.
2. Call `ToNumber(month)`.
3. If `date` is not specified, compute `DateFromTime(t)`, otherwise, call `ToNumber(date)`.
4. Compute `MakeDate(Result(4), TimeWithinDay(t))`.
5. Set the `[[PrimitiveValue]]` property of the `this` value to `TimeClip(Result(5))`.
6. Return the value of the `[[PrimitiveValue]]` property of the `this` value.

The `length` property of the `setMonth` method is 2.

15.9.5.39 `Date.prototype.setUTCMonth (month [, date ] )`
If `date` is not specified, this behaves as if `date` were specified with the value `getUTCDate()`.
1. Let `t` be this time value.
2. Call `ToNumber(month)`.
3. If `date` is not specified, compute `DateFromTime(t)`, otherwise, call `ToNumber(date)`.
4. Compute `MakeDate(Result(4), TimeWithinDay(t))`.
5. Set the `[[PrimitiveValue]]` property of the `this` value to `TimeClip(Result(5))`.
6. Return the value of the `[[PrimitiveValue]]` property of the `this` value.

The `length` property of the `setUTCMonth` method is 2.

15.9.5.40 `Date.prototype.setFullYear (year [, month [, date ] ] )`
If `month` is not specified, this behaves as if `month` were specified with the value `getMonth()`.
If `date` is not specified, this behaves as if `date` were specified with the value `getDate()`.
1. Let `t` be the result of `LocalTime(this time value)`; but if this time value is NaN, let `t` be +0.
2. Call `ToNumber(year)`.
3. If `month` is not specified, compute `MonthFromTime(t)`, otherwise, call `ToNumber(month)`.
4. If `date` is not specified, compute `DateFromTime(t)`, otherwise, call `ToNumber(date)`.
5. Compute `MakeDay(Result(2), Result(3), Result(4))`.
6. Compute `UTC(MakeDate(Result(5), TimeWithinDay(t)))`.
7. Set the `[[PrimitiveValue]]` property of the `this` value to `TimeClip(Result(6))`.
8. Return the value of the `[[PrimitiveValue]]` property of the `this` value.

The `length` property of the `setFullYear` method is 3.
15.9.5.41 `Date.prototype.setUTCFullYear(year, month[, date])`

If `month` is not specified, this behaves as if `month` were specified with the value `getUTCMonth()`.

If `date` is not specified, this behaves as if `date` were specified with the value `getUTCDate()`.

1. Let `t` be this time value; but if this time value is `NaN`, let `t` be `+0`.
2. Call `ToNumber(year)`.
3. If `month` is not specified, compute `MonthFromTime(t);` otherwise, call `ToNumber(month)`.
4. If `date` is not specified, compute `DateFromTime(t);` otherwise, call `ToNumber(date)`.
5. Compute `MakeDay(Result(2), Result(3), Result(4))`.
6. Compute `MakeDate(Result(5), TimeWithinDay(t))`.
7. Set the `[[PrimitiveValue]]` property of the `this` value to `TimeClip(Result(6))`.
8. Return the value of the `[[PrimitiveValue]]` property of the `this` value.

The `length` property of the `setUTCFullYear` method is 3.

15.9.5.42 `Date.prototype.toUTCString()`

This function returns a string value. The contents of the string are implementation-dependent, but are intended to represent the Date in a convenient, human-readable form in UTC.

NOTE: the intent is produce string representation of a date that is more readable than the format specified in Section 15.9.1.15. It is not essential that the chosen format be unambiguous or easily machine parsable. If an implementation does not have a preferred human-readable format it is recommended to use the format called out in Section 15.9.1.15 but with a space rather than a “-” used to separate the date and time elements.

15.9.5.43 `Date.prototype.toISOString()`

This function returns a string value. The format of the string is as called out in Date Time string format (15.9.1.15). All fields are present in the string. The time zone is always UTC, denoted by the suffix Z.

15.9.5.44 `Date.prototype.toJSON(key)`

This function returns the same string as `Date.prototype.toISOString()`.

When the `toJSON` method is called with argument `key`, the following steps are taken:

1. Let `O` be this object.
2. Call the `[[Get]]` method of `O` with argument “`toISOString`”.
3. If `IsCallable(Result(2))` is `false`, go to step 6.
4. Call the `[[Call]]` method of `Result(2)` with `O` as the `this` value and an empty argument list.
5. If `Result(4)` is a primitive value, return `Result(4)`.
6. Throw a `TypeError` exception.

NOTE

The `toJSON` function is intentionally generic; it does not require that its `this` value be a `Date` object. Therefore, it can be transferred to other kinds of objects for use as a method. An object is free to use the argument `key` that is passed in to filter its stringification.

15.9.6 Properties of Date Instances

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

15.10 RegExp (Regular Expression) Objects

A RegExp object contains a regular expression and the associated flags.

NOTE

The form and functionality of regular expressions is modelled after the regular expression facility in the Perl 5 programming language.

15.10.1 Patterns

The `RegExp` constructor applies the following grammar to the input pattern string. An error occurs if the grammar cannot interpret the string as an expansion of `Pattern`.

Syntax
Pattern ::
  Disjunction

Disjunction ::
  Alternative
  Alternative  |  Disjunction

Alternative ::
  [empty]
  Alternative Term

Term ::
  Assertion
  Atom
  Atom Quantifier

Assertion ::
  ^
  \b
  \B

Quantifier ::
  QuantifierPrefix
  QuantifierPrefix ?

QuantifierPrefix ::
  *
  +
  ?
  { DecimalDigits }
  { DecimalDigits , }
  { DecimalDigits , DecimalDigits }

Atom ::
  PatternCharacter
  -
  \ AtomEscape
  CharacterClass
    { Disjunction }
    { ? : Disjunction }
    { ? = Disjunction }
    { ? ! Disjunction }

PatternCharacter :: SourceCharacter but not any of:
  ^  $  \  .  *  +  ?  (  )  [  ]  {  }  |

AtomEscape ::
  DecimalEscape
  CharacterEscape
  CharacterClassEscape

CharacterEscape ::
  ControlEscape
  ControlLetter
  HexEscapeSequence
  UnicodeEscapeSequence
  IdentityEscape

15 December 2008
ControlEscape :: one of
    f n r t v

ControlLetter :: one of
    a b c d e f g h i j k l m n o p q r s t u v w x y z
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

IdentityEscape ::
    SourceCharacter but not IdentifierPart

DecimalEscape ::
    DecimalIntegerLiteral [lookahead a DecimalDigit]

CharacterClassEscape :: one of
    d D s S w W

CharacterClass ::
    [ [lookahead a (* ) ClassRanges ]
     [ ^ ClassRanges ]

ClassRanges ::
    (empty)
    NonemptyClassRanges

NonemptyClassRanges ::
    ClassAtom
    ClassAtom NonemptyClassRangesNoDash
    ClassAtom - ClassAtom ClassRanges

NonemptyClassRangesNoDash ::
    ClassAtom
    ClassAtomNoDash NonemptyClassRangesNoDash
    ClassAtomNoDash - ClassAtom ClassRanges

ClassAtom ::
    -
    ClassAtomNoDash

ClassAtomNoDash ::
    SourceCharacter but not one of \ ] -
    \ ClassEscape

ClassEscape ::
    DecimalEscape
    b
    CharacterEscape
    CharacterClassEscape

15.10.2 Pattern Semantics

A regular expression pattern is converted into an internal procedure using the process described below. An implementation is encouraged to use more efficient algorithms than the ones listed below, as long as the results are the same. The internal procedure is used as the value of a RegExp object’s [[Match]] internal property.

15.10.2.1 Notation

The descriptions below use the following variables:

Input is the string being matched by the regular expression pattern. The notation input[n] means the n-th character of input, where n can range between 0 (inclusive) and InputLength (exclusive).
InputLength is the number of characters in the Input string.

NcapturingParens, is the total number of left capturing parentheses (i.e. the total number of times the Atom :: ( Disjunction ) production is expanded) in the pattern. A left capturing parenthesis is any { pattern character that is matched by the { terminal of the Atom :: ( Disjunction ) production.

IgnoreCase is the setting of the RegExp object’s ignoreCase property.

Multiline is the setting of the RegExp object’s multiline property.

Furthermore, the descriptions below use the following internal data structures:

A CharSet is a mathematical set of characters.

A State is an ordered pair ( endIndex, captures) where endIndex is an integer and captures is an internal array of NcapturingParens values. States are used to represent partial match states in the regular expression matching algorithms. The endIndex is one plus the index of the last input character matched so far by the pattern, while captures holds the results of capturing parentheses. The nth element of captures is either a string that represents the value obtained by the nth set of capturing parentheses or undefined if the nth set of capturing parentheses hasn’t been reached yet. Due to backtracking, many states may be in use at any time during the matching process.

A MatchResult is either a State or the special token failure that indicates that the match failed.

A Continuation procedure is an internal closure (i.e. an internal procedure with some arguments already bound to values) that takes one State argument and returns a MatchResult result. If an internal closure references variables bound in the function that creates the closure, the closure uses the values that these variables had at the time the closure was created. The continuation attempts to match the remaining portion (specified by the closure’s already-bound arguments) of the pattern against the input string, starting at the intermediate state given by its State argument. If the match succeeds, the continuation returns the final State that it reached; if the match fails, the continuation returns failure.

A Matcher procedure is an internal closure that takes two arguments -- a State and a Continuation -- and returns a MatchResult result. The matcher attempts to match a middle subpattern (specified by the closure’s already-bound arguments) of the pattern against the input string, starting at the intermediate state given by its State argument. The Continuation argument should be a closure that matches the rest of the pattern. After matching the subpattern of a pattern to obtain a new State, the matcher then calls Continuation on that state to test if the rest of the pattern can match as well. If it can, the matcher returns the state returned by the continuation; if not, the matcher may try different choices at its choice points, repeatedly calling Continuation until it either succeeds or all possibilities have been exhausted.

An AssertionTester procedure is an internal closure that takes a State argument and returns a boolean result. The assertion tester tests a specific condition (specified by the closure’s already-bound arguments) against the current place in the input string and returns true if the condition matched or false if not.

An EscapeValue is either a character or an integer. An EscapeValue is used to denote the interpretation of a DecimalEscape escape sequence: a character ch means that the escape sequence is interpreted as the character ch, while an integer n means that the escape sequence is interpreted as a backreference to the nth set of capturing parentheses.

15.10.2.2 Pattern

The production Pattern :: Disjunction evaluates as follows:

1. Evaluate Disjunction to obtain a Matcher m.
2. Return an internal closure that takes two arguments, a string str and an integer index, and performs the following:
   1. Let Input be the given string str. This variable will be used throughout the algorithms in 15.10.2.
   2. Let InputLength be the length of Input. This variable will be used throughout the algorithms in 15.10.2.
   3. Let c be a Continuation that always returns its State argument as a successful MatchResult.
   4. Let cap be an internal array of NcapturingParens undefined values, indexed 1 through NcapturingParens.
   5. Let x be the State (index, cap).

15 December 2008
6. Call \( m(x, c) \) and return its result.

**Informative comments:** A Pattern evaluates ("compiles") to an internal **procedure** value. RegExp.prototype.exec can then apply this **procedure** to a string and an offset within the string to determine whether the pattern would match starting at exactly that offset within the string, and, if it does match, what the values of the capturing parentheses would be. The algorithms in 15.10.2 are designed so that compiling a pattern may throw a SyntaxError exception; on the other hand, once the pattern is successfully compiled, applying its result **internal procedure** to find a match in a string cannot throw an exception (except for any host-defined exceptions that can occur anywhere such as out-of-memory).

15.10.2.3 **Disjunction**

The production Disjunction :: Alternative evaluates by evaluating Alternative to obtain a Matcher and returning that Matcher.

The production Disjunction :: Alternative | Disjunction evaluates as follows:

1. Evaluate Alternative to obtain a Matcher \( m_1 \).
2. Evaluate Disjunction to obtain a Matcher \( m_2 \).
3. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following:
   1. Call \( m_1(x, c) \) and let \( r \) be its result.
   2. If \( r \) isn’t failure, return \( r \).
   3. Call \( m_2(x, c) \) and return its result.

**Informative comments:** The \(|\) regular expression operator separates two alternatives. The pattern first tries to match the left Alternative (followed by the sequel of the regular expression); if it fails, it tries to match the right Disjunction (followed by the sequel of the regular expression). If the left Alternative, the right Disjunction, and the sequel all have choice points, all choices in the sequel are tried before moving on to the next choice in the left Alternative. If choices in the left Alternative are exhausted, the right Disjunction is tried instead of the left Alternative. Any capturing parentheses inside a portion of the pattern skipped by \(|\) produce undefined values instead of strings. Thus, for example,

\[
/a|ab/.exec("abc")
\]

returns the result "a" and not "ab". Moreover,

\[
/(\{(a)\|(ab)\}\{(c)\|(bc)\})/.exec("abc")
\]

returns the array

\[ ["abc", "a", "ab", undefined, "bc", undefined, "bc"] \]

and not

\[ ["abc", "ab", undefined, "ab", "c", "c", undefined] \]

15.10.2.4 **Alternative**

The production Alternative :: [empty] evaluates by returning a Matcher that takes two arguments, a State \( x \) and a Continuation \( c \), and returns the result of calling \( c(x) \).

The production Alternative :: Alternative Term evaluates as follows:

1. Evaluate Alternative to obtain a Matcher \( m_1 \).
2. Evaluate Term to obtain a Matcher \( m_2 \).
3. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following:
   1. Create a Continuation \( d \) that takes a State argument \( y \) and returns the result of calling \( m_2(y, c) \).
   2. Call \( m_1(x, d) \) and return its result.

**Informative comments:** Consecutive Terms try to simultaneously match consecutive portions of the input string. If the left Alternative, the right Term, and the sequel of the regular expression all have
choice points, all choices in the sequel are tried before moving on to the next choice in the right Term, and all choices in the right Term are tried before moving on to the next choice in the left Alternative.

15.10.2.5 Term

The production Term :: Assertion evaluates by returning an internal Matcher closure that takes two arguments, a State x and a Continuation c, and performs the following:

1. Evaluate Assertion to obtain an AssertionTester t.
2. Call t(s) and let r be the resulting boolean value.
3. If r is false, return failure.
4. Call c(x) and return its result.

The production Term :: Atom evaluates by evaluating Atom to obtain a Matcher and returning that Matcher.

The production Term :: Atom Quantifier evaluates as follows:

1. Evaluate Atom to obtain a Matcher m.
2. Evaluate Quantifier to obtain the three results: an integer min, an integer (or \( \infty \)) max, and boolean greedy.
3. If max is finite and less than min, then throw a SyntaxError exception.
4. Let parenIndex be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion’s Term. This is the total number of times the Atom :: ( Disjunction ) production is expanded prior to this production’s Term plus the total number of Atom :: ( Disjunction ) productions enclosing this Term.
5. Let parenCount be the number of left capturing parentheses in the expansion of this production’s Atom. This is the total number of Atom :: ( Disjunction ) productions enclosed by this production’s Atom.
6. Return an internal Matcher closure that takes two arguments, a State x and a Continuation c, and performs the following:
   1. Call RepeatMatcher(m, min, max, greedy, x, c, parenIndex, parenCount) and return its result.

The abstract operation RepeatMatcher takes eight parameters, a Matcher m, an integer min, an integer (or \( \infty \)) max, a boolean greedy, a State x, a Continuation c, an integer parenIndex, and an integer parenCount, and performs the following:

1. If max is zero, then call c(x) and return its result.
2. Create an internal Continuation closure d that takes one State argument y and performs the following:
   1. If min is zero and y’s endIndex is equal to x’s endIndex, then return failure.
   2. If min is zero then let min2 be zero; otherwise let min2 be min–1.
   3. If max is \( \infty \), then let max2 be \( \infty \); otherwise let max2 be max–1.
   4. Call RepeatMatcher(m, min2, max2, greedy, y, c, parenIndex, parenCount) and return its result.
3. Let cap be a fresh copy of x’s captures internal array.
4. For every integer k that satisfies parenIndex + k < parenIndex+parenCount, set cap[k] to undefined.
5. Let e be x’s endIndex.
6. Let xr be the Stație (e, cap).
7. If min is not zero, then call m(xr, d) and return its result.
8. If greedy is true, then go to step 12.
9. Call c(x) and let z be its result.
10. If z is not failure, return z.
11. Call m(xr, d) and return its result.
12. Call m(xr, d) and let z be its result.
13. If z is not failure, return z.
14. Call c(x) and return its result.

Informative comments: An Atom followed by a Quantifier is repeated the number of times specified by the Quantifier. A quantifier can be non-greedy, in which case the Atom pattern is repeated as few times as possible while still matching the sequel, or it can be greedy, in which case the Atom pattern is
repeated as many times as possible while still matching the sequel. The Atom pattern is repeated rather than the input string that it matches, so different repetitions of the Atom can match different input substrings.

If the Atom and the sequel of the regular expression all have choice points, the Atom is first matched as many (or as few, if non-greedy) times as possible. All choices in the sequel are tried before moving on to the next choice in the last repetition of Atom. All choices in the last (n-th) repetition of Atom are tried before moving on to the next choice in the next-to-last (n-1)th repetition of Atom; at which point it may turn out that more or fewer repetitions of Atom are now possible; these are exhausted (again, starting with either as few or as many as possible) before moving on to the next choice in the (n-1)th repetition of Atom and so on.

Consider also

```
/\{aa|aabac|ba|b|c\}\*/.exec("aabac")
```

which, by the choice point ordering above, returns the array

```
["aabac", "ba"]
```

and not any of:

```
["aabac", "aabac"]
["aabac", "c"]
```

The above ordering of choice points can be used to write a regular expression that calculates the greatest common divisor of two numbers (represented in unary notation). The following example calculates the gcd of 10 and 15:

```
"aaaaaaaaaa,aaaaaaaaaaaaaaa".replace(/(^\{a+\}\1*,\1+$/,"$1")
```

which returns the gcd in unary notation "aaaaa".

Step 4 of the RepeatMatcher clears Atom’s captures each time Atom is repeated. We can see its behaviour in the regular expression

```
/\{z\}(a+)?(b+)?(c)\*/.exec("zaacbbbcac")
```

which returns the array

```
["zaacbbbcac", "z", "ac", "a", undefined, "c"]
```

and not

```
["zaacbbbcac", "z", "ac", "a", "bbb", "c"]
```

because each iteration of the outermost * clears all captured strings contained in the quantified Atom, which in this case includes capture strings numbered 2, 3, and 4.

Step 1 of the RepeatMatcher’s closure d states that, once the minimum number of repetitions has been satisfied, any more expansions of Atom that match the empty string are not considered for further repetitions. This prevents the regular expression engine from falling into an infinite loop on patterns such as:

```
/(a*)\*/.exec("b")
```

or the slightly more complicated:

```
/(a*)b\1+/.exec("baaaac")
```

which returns the array

15 December 2008
15.10.2.6 Assertion

The production `Assertion :: ^` evaluates by returning an internal `AssertionTester` closure that takes a State argument \( x \) and performs the following:

1. Let \( e \) be \( x \)'s endIndex.
2. If \( e \) is zero, return `true`.
3. If `Multiline` is `false`, return `false`.
4. If the character `Input[e-1]` is one of `LineTerminator`, return `true`.
5. Return `false`.

The production `Assertion :: $` evaluates by returning an internal `AssertionTester` closure that takes a State argument \( x \) and performs the following:

1. Let \( e \) be \( x \)'s endIndex.
2. If \( e \) is equal to `InputLength`, return `true`.
3. If `multiline` is `false`, return `false`.
4. If the character `Input[e]` is one of `LineTerminator`, return `true`.
5. Return `false`.

The production `Assertion :: b` evaluates by returning an internal `AssertionTester` closure that takes a State argument \( x \) and performs the following:

1. Let \( e \) be \( x \)'s endIndex.
2. Call `IsWordChar(e-1)` and let \( a \) be the boolean result.
3. Call `IsWordChar(e)` and let \( b \) be the boolean result.
4. If \( a \) is `true` and \( b \) is `false`, return `true`.
5. If \( a \) is `false` and \( b \) is `true`, return `true`.
6. Return `false`.

The abstract operation `IsWordChar` takes an integer parameter \( e \) and performs the following:

1. If \( e == -1 \) or \( e == \text{InputLength} \), return `false`.
2. Let \( c \) be the character `Input[e]`.
3. If \( c \) is one of the sixty-three characters in the table below, return `true`.

| A | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | _ |

4. Return `false`.

15.10.2.7 Quantifier

The production `Quantifier :: QuantifierPrefix` evaluates as follows:

1. Evaluate `QuantifierPrefix` to obtain the two results: an integer `min` and an integer (or \( \infty \)) `max`.
2. Return the three results `min`, `max`, and `true`.

The production `Quantifier :: QuantifierPrefix ?` evaluates as follows:
1. Evaluate \textit{QuantifierPrefix} to obtain the two results: an integer \textit{min} and an integer (or \textit{max}) \textit{max}.
2. Return the three results \textit{min}, \textit{max}, and \textit{false}.

The production \textit{QuantifierPrefix} :: * evaluates by returning the two results 0 and \textit{max}.

The production \textit{QuantifierPrefix} :: + evaluates by returning the two results 1 and \textit{max}.

The production \textit{QuantifierPrefix} :: ? evaluates by returning the two results 0 and 1.

The production \textit{QuantifierPrefix} :: \{ \textit{DecimalDigits} \} evaluates as follows:

1. Let \textit{i} be the MV of \textit{DecimalDigits} (see 7.8.3).
2. Return the two results \textit{i} and \textit{i}.

The production \textit{QuantifierPrefix} :: \{ \textit{DecimalDigits} , \textit{DecimalDigits} \} evaluates as follows:

1. Let \textit{i} be the MV of the first \textit{DecimalDigits}.
2. Let \textit{j} be the MV of the second \textit{DecimalDigits}.
3. Return the two results \textit{i} and \textit{j}.

15.10.2.8 \textit{Atom}

The production \textit{Atom} :: \textit{PatternCharacter} evaluates as follows:

1. Let \textit{ch} be the character represented by \textit{PatternCharacter}.
2. Let \textit{A} be a one-element CharSet containing the character \textit{ch}.
3. Call \textit{CharacterSetMatcher(A, false)} and return its Matcher result.

The production \textit{Atom} :: . evaluates as follows:

1. Let \textit{A} be the set of all characters except \textit{LineTerminator}.
2. Call \textit{CharacterSetMatcher(A, false)} and return its Matcher result.

The production \textit{Atom} :: \textit{AtomEscape} evaluates by evaluating \textit{AtomEscape} to obtain a Matcher and returning that Matcher.

The production \textit{Atom} :: \textit{CharacterClass} evaluates as follows:

1. Evaluate \textit{CharacterClass} to obtain a CharSet \textit{A} and a boolean \textit{invert}.

The production \textit{Atom} :: \{ \textit{Disjunction} \} evaluates as follows:

1. Evaluate \textit{Disjunction} to obtain a Matcher \textit{m}.
2. Let \textit{parenIndex} be the number of left capturing parentheses in the entire regular expression that occur to the left of this production expansion's initial left parenthesis. This is the total number of times the \textit{Atom} :: \{ \textit{Disjunction} \} production is expanded prior to this production's \textit{Atom} plus the total number of \textit{Atom} :: \{ \textit{Disjunction} \} productions enclosing this \textit{Atom}.
3. Return an internal Matcher closure that takes two arguments, a State \textit{x} and a Continuation \textit{c}, and performs the following:

1. Create an internal Continuation closure \textit{d} that takes one State argument \textit{y} and performs the following:

   - Let \textit{cap} be a fresh copy of \textit{y}'s \textit{captures} internal array.
   - Let \textit{xe} be \textit{x}'s endIndex.
   - Let \textit{ye} be \textit{y}'s endIndex.
   - Let \textit{s} be a fresh string whose characters are the characters of \textit{Input} at positions \textit{xe} (inclusive) through \textit{ye} (exclusive).
   - Set \textit{cap[parenIndex+1]} to \textit{s}.
   - Let \textit{z} be the State (\textit{ye}, \textit{cap}).

15 December 2008
Call \( c(z) \) and return its result.
2. Call \( m(x, d) \) and return its result.

The production \( \text{Atom} :: ( ~ ? ~ : \text{Disjunction} ~ ) \) evaluates by evaluating \( \text{Disjunction} \) to obtain a Matcher and returning that Matcher.

The production \( \text{Atom} :: ( ~ ? = \text{Disjunction} ~ ) \) evaluates as follows:
1. Evaluate \( \text{Disjunction} \) to obtain a Matcher \( m \).
2. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following:
   1. Let \( d \) be a Continuation that always returns its State argument as a successful MatchResult.
   2. Call \( m(x, d) \) and let \( r \) be its result.
   3. If \( r \) is failure, return failure.
   4. Let \( y \) be \( r \)'s State.
   5. Let \( c \) be \( y \)'s captures internal array.
   6. Let \( z \) be \( y \)'s \( \text{endIndex} \).
   7. Call \( c(z) \) and return its result.

The production \( \text{Atom} :: ( ~ ? ! \text{Disjunction} ~ ) \) evaluates as follows:
1. Evaluate \( \text{Disjunction} \) to obtain a Matcher \( m \).
2. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following:
   1. Let \( d \) be a Continuation that always returns its State argument as a successful MatchResult.
   2. Call \( m(x, d) \) and let \( r \) be its result.
   3. If \( r \) isn't failure, return failure.
   4. Call \( c(x) \) and return its result.

The \textit{abstract operation} \texttt{CharacterSetMatcher} takes two arguments, a CharSet \( A \) and a boolean flag \texttt{invert}, and performs the following:
1. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following:
   1. Let \( e \) be \( x \)'s \( \text{endIndex} \).
   2. If \( e == \text{InputLength} \), return failure.
   3. Let \( c \) be the character \( \text{Input}[e] \).
   4. Let \( cc \) be the result of \texttt{Canonicalize}(\( c \)).
   5. If \texttt{invert} is true, go to step 8.
   6. If there does not exist a member \( a \) of set \( A \) such that \texttt{Canonicalize}(\( a \)) == \( cc \), then return failure.
   8. If there exists a member \( a \) of set \( A \) such that \texttt{Canonicalize}(\( a \)) == \( cc \), then return failure.
   9. Let \( cap \) be \( y \)'s captures internal array.
   10. Let \( y \) be the State (\( x+1 \), \( cap \)).
   11. Call \( c(y) \) and return its result.

The \textit{abstract operation} \texttt{Canonicalize} takes a character parameter \( ch \) and performs the following:
1. If \texttt{IgnoreCase} is false, return \( ch \).
2. Let \( u \) be \( ch \) converted to upper case as if by calling the standard built-in method \texttt{String.prototype.toUpperCase} on the one-character string \( ch \).
3. If \( u \) does not consist of a single character, return \( ch \).
4. Let \( cu \) be \( u \)'s character.
5. If \( ch \)'s code point value is greater than or equal to decimal 128 and \( cu \)'s code point value is less than decimal 128, then return \( ch \).
6. Return \( cu \).

\textbf{Informative comments:} Parentheses of the form \( (\text{Disjunction}~) \) serve both to group the components of the \texttt{Disjunction} pattern together and to save the result of the match. The result can be
used either in a backreference (\ followed by a nonzero decimal number), referenced in a replace string, or returned as part of an array from the regular expression matching. To inhibit the capturing behaviour of parentheses, use the form (? : Disjunction ) instead.

The form (?= Disjunction ) specifies a zero-width positive lookahead. In order for it to succeed, the pattern inside Disjunction must match at the current position, but the current position is not advanced before matching the sequel. If Disjunction can match at the current position in several ways, only the first one is tried. Unlike other regular expression operators, there is no backtracking into a (?= form (this unusual behaviour is inherited from Perl). This only matters when the Disjunction contains capturing parentheses and the sequel of the pattern contains backreferences to those captures.

For example,

```
/(?=(a+))/.exec("baaabac")
```

matches the empty string immediately after the first b and therefore returns the array:

```
["", "aaa"]
```

To illustrate the lack of backtracking into the lookahead, consider:

```
/(?=(a+))a*b\1/.exec("baaabac")
```

This expression returns

```
["aba", "a"]
```

and not:

```
["aaaba", "a"]
```

The form (?! Disjunction ) specifies a zero-width negative lookahead. In order for it to succeed, the pattern inside Disjunction must fail to match at the current position. The current position is not advanced before matching the sequel. Disjunction can contain capturing parentheses, but backreferences to them only make sense from within Disjunction itself. Backreferences to these capturing parentheses from elsewhere in the pattern always return undefined because the negative lookahead must fail for the pattern to succeed. For example,

```
/\(?! (a+)\b\2c)\2(.*)/.exec("baaabac")
```

looks for an a not immediately followed by some positive number n of a's, a b, another n a's (specified by the first \2) and a c. The second \2 is outside the negative lookahead, so it matches against undefined and therefore always succeeds. The whole expression returns the array:

```
["baaabac", "ba", undefined, "abaac"]
```

In case-insensitive matches all characters are implicitly converted to upper case immediately before they are compared. However, if converting a character to upper case would expand that character into more than one character (such as converting "ß" (\u00DF) into "SS"), then the character is left as-is instead. The character is also left as-is if it is not an ASCII character but converting it to upper case would make it into an ASCII character. This prevents Unicode characters such as \u0131 and \u017F from matching regular expressions such as /[^\w]/i, which are only intended to match ASCII letters. Furthermore, if these conversions were allowed, then /[^\w]/i would match each of a, b, …, h, but not i or s.

15.10.2.9 AtomEscape

The production AtomEscape :: DecimalEscape evaluates as follows:

1. Evaluate DecimalEscape to obtain an EscapeValue E.
2. If E is not a character then go to step 6.
3. Let ch be E's character.
4. Let A be a one-element CharSet containing the character ch.
5. Call CharacterSetMatcher(A, false) and return its Matcher result.
6. E must be an integer. Let n be that integer.
7. If n=0 or n>NCapturingParens then throw a SyntaxError exception.
8. Return an internal Matcher closure that takes two arguments, a State \( x \) and a Continuation \( c \), and performs the following:
   1. Let \( \text{cap} \) be \( x \)'s captures internal array.
   2. Let \( s \) be \( \text{cap}[^n] \).
   3. If \( s \) is \text{undefined}, then call \( c(x) \) and return its result.
   4. Let \( e \) be \( x \)'s endIndex.
   5. Let \( \text{len} \) be \( s \)'s length.
   6. Let \( f \) be \( e + \text{len} \).
   7. If \( f > \text{InputLength} \), return \text{failure}.
   8. If there exists an integer \( i \) between 0 (inclusive) and \( \text{len} \) (exclusive) such that
      \( \text{Canonicalize}(s[i]) \) is not the same character as
      \( \text{Canonicalize(Input}[e+i]) \), then return \text{failure}.
   9. Let \( y \) be the State \( (f, \text{cap}) \).
   10. Call \( c(y) \) and return its result.

The production \( \text{AtomEscape} :: \text{CharacterEscape} \) evaluates as follows:
   1. Evaluate \( \text{CharacterEscape} \) to obtain a character \( ch \).
   2. Let \( A \) be a one-element CharSet containing the character \( ch \).
   3. Call \( \text{CharacterSetMatcher}(A, \text{false}) \) and return its Matcher result.

The production \( \text{AtomEscape} :: \text{CharacterClassEscape} \) evaluates as follows:
   1. Evaluate \( \text{CharacterClassEscape} \) to obtain a CharSet \( A \).
   2. Call \( \text{CharacterSetMatcher}(A, \text{false}) \) and return its Matcher result.

Informative comments:
An escape sequence of the form \( \backslash \) followed by a nonzero decimal number \( n \)
matches the result of the \( n \)th set of capturing parentheses (see 15.10.2.11). It is an error if the regular
expression has fewer than \( n \) capturing parentheses. If the regular expression has \( n \) or more capturing
parentheses but the \( n \)th one is \text{undefined} because it hasn’t captured anything, then the backreference
always succeeds.

### 15.10.2.10 CharacterEscape

The production \( \text{CharacterEscape} :: \text{ControlEscape} \) evaluates by returning the character according to
the table below:

<table>
<thead>
<tr>
<th>ControlEscape</th>
<th>Unicode Value</th>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>\u0009</td>
<td>horizontal tab</td>
<td>&lt;HT&gt;</td>
</tr>
<tr>
<td>n</td>
<td>\u000A</td>
<td>line feed (new line)</td>
<td>&lt;LF&gt;</td>
</tr>
<tr>
<td>v</td>
<td>\u000B</td>
<td>vertical tab</td>
<td>&lt;VT&gt;</td>
</tr>
<tr>
<td>f</td>
<td>\u000C</td>
<td>form feed</td>
<td>&lt;FF&gt;</td>
</tr>
<tr>
<td>r</td>
<td>\u000D</td>
<td>carriage return</td>
<td>&lt;CR&gt;</td>
</tr>
</tbody>
</table>

The production \( \text{CharacterEscape} :: \text{ControlLetter} \) evaluates as follows:
   1. Let \( ch \) be the character represented by \text{ControlLetter}.
   2. Let \( i \) be \( ch \)'s code point value.
   3. Let \( j \) be the remainder of dividing \( i \) by 32.
   4. Return the Unicode character numbered \( j \).

The production \( \text{CharacterEscape} :: \text{HexEscapeSequence} \) evaluates by evaluating the CV of the
\( \text{HexEscapeSequence} \) (see 7.8.4) and returning its character result.

The production \( \text{CharacterEscape} :: \text{UnicodeEscapeSequence} \) evaluates by evaluating the CV of the
\( \text{UnicodeEscapeSequence} \) (see 7.8.4) and returning its character result.

The production \( \text{CharacterEscape} :: \text{IdentityEscape} \) evaluates by returning the character represented by
\text{IdentityEscape}.

### 15.10.2.11 DecimalEscape

The production \( \text{DecimalEscape} :: \text{DecimalIntegerLiteral} \) evaluates as follows.
1. Let $i$ be the MV of $\text{DecimalIntegerLiteral}$.
2. If $i$ is zero, return the EscapeValue consisting of a $<\text{NUL}>$ character (Unicode value 0000).
3. Return the EscapeValue consisting of the integer $i$.

The definition of “the MV of $\text{DecimalIntegerLiteral}$” is in 7.8.3.

Informative comments: If \ is followed by a decimal number $n$ whose first digit is not 0, then the escape sequence is considered to be a backreference. It is an error if $n$ is greater than the total number of left capturing parentheses in the entire regular expression. \0 represents the NUL character and cannot be followed by a decimal digit.

15.10.2.12 CharacterClassEscape

The production $\text{CharacterClassEscape} :: d$ evaluates by returning the ten-element set of characters containing the characters 0 through 9 inclusive.

The production $\text{CharacterClassEscape} :: D$ evaluates by returning the set of all characters not included in the set returned by $\text{CharacterClassEscape} :: d$.

The production $\text{CharacterClassEscape} :: s$ evaluates by returning the set of characters containing the characters that are on the right-hand side of the $\text{WhiteSpace}$ (7.2) or $\text{LineTerminator}$ (7.3) productions.

The production $\text{CharacterClassEscape} :: S$ evaluates by returning the set of all characters not included in the set returned by $\text{CharacterClassEscape} :: s$.

The production $\text{CharacterClassEscape} :: w$ evaluates by returning the set of characters containing the sixty-three characters:

```
 a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
```

The production $\text{CharacterClassEscape} :: W$ evaluates by returning the set of all characters not included in the set returned by $\text{CharacterClassEscape} :: w$.

15.10.2.13 CharacterClass

The production $\text{CharacterClass} :: \{ \text{lookahead} ^ {\text{ClassRanges}} \} \}$ evaluates by evaluating $\text{ClassRanges}$ to obtain a CharSet and returning that CharSet and the boolean false.

The production $\text{CharacterClass} :: \{ ^ {\text{ClassRanges}} \}$ evaluates by evaluating $\text{ClassRanges}$ to obtain a CharSet and returning that CharSet and the boolean true.

15.10.2.14 ClassRanges

The production $\text{ClassRanges} :: \emptyset$ evaluates by returning the empty CharSet.

The production $\text{ClassRanges} :: \{ \text{NonemptyClassRanges} \}$ evaluates by evaluating $\text{NonemptyClassRanges}$ to obtain a CharSet and returning that CharSet.

15.10.2.15 NonemptyClassRanges

The production $\text{NonemptyClassRanges} :: \text{ClassAtom}$ evaluates by evaluating $\text{ClassAtom}$ to obtain a CharSet and returning that CharSet.

The production $\text{NonemptyClassRanges} :: \text{ClassAtom} \text{NonemptyClassRangesNoDash}$ evaluates as follows:

1. Evaluate $\text{ClassAtom}$ to obtain a CharSet $A$.
2. Evaluate $\text{NonemptyClassRangesNoDash}$ to obtain a CharSet $B$.
3. Return the union of CharSets $A$ and $B$.

The production $\text{NonemptyClassRanges} :: \text{ClassAtom} \text{ClassRanges}$ evaluates as follows:

1. Evaluate the first $\text{ClassAtom}$ to obtain a CharSet $A$.
2. Evaluate the second $\text{ClassAtom}$ to obtain a CharSet $B$.
3. Evaluate $\text{ClassRanges}$ to obtain a CharSet $C$.
4. Call `CharacterRange(A, B)` and let `D` be the resulting CharSet.
5. Return the union of CharSets `D` and `C`.

The abstract operation `CharacterRange` takes two CharSet parameters `A` and `B` and performs the following:

1. If `A` does not contain exactly one character or `B` does not contain exactly one character then throw a `SyntaxError` exception.
2. Let `a` be the one character in CharSet `A`.
3. Let `b` be the one character in CharSet `B`.
4. Let `i` be the code point value of character `a`.
5. Let `j` be the code point value of character `b`.
6. If `i > j` then throw a `SyntaxError` exception.
7. Return the set containing all characters numbered `i` through `j`, inclusive.

15.10.2.16 NonemptyClassRangesNoDash

The production `NonemptyClassRangesNoDash ::= ClassAtom` evaluates by evaluating `ClassAtom` to obtain a CharSet and returning that CharSet.

The production `NonemptyClassRangesNoDash ::= ClassAtomNoDash NonemptyClassRangesNoDash` evaluates as follows:

1. Evaluate `ClassAtomNoDash` to obtain a CharSet `A`.
2. Evaluate `NonemptyClassRangesNoDash` to obtain a CharSet `B`.
3. Return the union of CharSets `A` and `B`.

The production `NonemptyClassRangesNoDash ::= ClassAtomNoDash - ClassAtom ClassRanges` evaluates as follows:

1. Evaluate `ClassAtomNoDash` to obtain a CharSet `A`.
2. Evaluate `ClassAtom` to obtain a CharSet `B`.
3. Evaluate `ClassRanges` to obtain a CharSet `C`.
4. Call `CharacterRange(A, B)` and let `D` be the resulting CharSet.
5. Return the union of CharSets `D` and `C`.

Informative comments: `ClassRanges` can expand into single `ClassAtoms` and/or ranges of two `ClassAtoms` separated by dashes. In the latter case the `ClassRanges` includes all characters between the first `ClassAtom` and the second `ClassAtom`, inclusive; an error occurs if either `ClassAtom` does not represent a single character (for example, if one is `\w`) or if the first `ClassAtom`'s code point value is greater than the second `ClassAtom`'s code point value.

Even if the pattern ignores case, the case of the two ends of a range is significant in determining which characters belong to the range. Thus, for example, the pattern `/[E-F]/i` matches only the letters `E`, `F`, `e`, and `f`, while the pattern `/[E-F]/i` matches all upper and lower-case ASCII letters as well as the symbols `[`, `\`, `^`, `_`, and `. .

A `–` character can be treated literally or it can denote a range. It is treated literally if it is the first or last character of `ClassRanges`, the beginning or end limit of a range specification, or immediately follows a range specification.

15.10.2.17 ClassAtom

The production `ClassAtom ::= -` evaluates by returning the CharSet containing the one character `-`.

The production `ClassAtom ::= ClassAtomNoDash` evaluates by evaluating `ClassAtomNoDash` to obtain a CharSet and returning that CharSet.

15.10.2.18 ClassAtomNoDash

The production `ClassAtomNoDash ::= SourceCharacter but not one of \` -` evaluates by returning a one-element CharSet containing the character represented by `SourceCharacter`.

The production `ClassAtomNoDash ::= \` ClassEscape` evaluates by evaluating `ClassEscape` to obtain a CharSet and returning that CharSet.
15.10.2.19 ClassEscape

The production \texttt{ClassEscape :: DecimalEscape} evaluates as follows:

1. Evaluate \texttt{DecimalEscape} to obtain an \texttt{EscapeValue} \texttt{E}.
2. If \texttt{E} is not a character then throw a \texttt{SyntaxError} exception.
3. Let \texttt{ch} be \texttt{E}'s character.
4. Return the one-element CharSet containing the character \texttt{ch}.

The production \texttt{ClassEscape :: b} evaluates by returning the CharSet containing the one character \texttt{<BS>} (Unicode value 0008).

The production \texttt{ClassEscape :: CharacterEscape} evaluates by evaluating \texttt{CharacterEscape} to obtain a character and returning a one-element CharSet containing that character.

The production \texttt{ClassEscape :: CharacterClassEscape} evaluates by evaluating \texttt{CharacterClassEscape} to obtain a CharSet and returning that CharSet.

Informative comments: A \texttt{ClassAtom} can use any of the escape sequences that are allowed in the rest of the regular expression except for \texttt{\b}, \texttt{\B}, and backreferences. Inside a \texttt{CharacterClass}, \texttt{\b} means the backspace character, while \texttt{\B} and backreferences raise errors. Using a backreference inside a \texttt{ClassAtom} causes an error.

15.10.3 The RegExp Constructor Called as a Function

15.10.3.1 \texttt{RegExp(pattern, flags)}

If \texttt{pattern} is an object \texttt{R} whose \texttt{[[Class]]} property is \texttt{"RegExp"} and \texttt{flags} is \texttt{undefined}, then return \texttt{R} unchanged. Otherwise call the standard built-in RegExp constructor (15.10.4.1), passing it the \texttt{pattern} and \texttt{flags} arguments and return the object constructed by that constructor.

15.10.4 The RegExp Constructor

When \texttt{RegExp} is called as part of a \texttt{new} expression, it is a constructor: it initialises the newly created object.

15.10.4.1 \texttt{new RegExp(pattern, flags)}

If \texttt{pattern} is an object \texttt{R} whose \texttt{[[Class]]} property is \texttt{"RegExp"} and \texttt{flags} is \texttt{undefined}, then let \texttt{P} be the \texttt{pattern} used to construct \texttt{R} and let \texttt{F} be the \texttt{flags} used to construct \texttt{R}. If \texttt{pattern} is an object \texttt{R} whose \texttt{[[Class]]} property is \texttt{"RegExp"} and \texttt{flags} is not \texttt{undefined}, then throw a \texttt{TypeError} exception. Otherwise, let \texttt{P} be the empty string if \texttt{pattern} is \texttt{undefined} and \texttt{ToString(pattern)} otherwise, and let \texttt{F} be the empty string if \texttt{flags} is \texttt{undefined} and \texttt{ToString(flags)} otherwise.

The \texttt{global} property of the newly constructed object is set to a Boolean value that is \texttt{true} if \texttt{F} contains the character \texttt{"g"} and \texttt{false} otherwise.

The \texttt{ignoreCase} property of the newly constructed object is set to a Boolean value that is \texttt{true} if \texttt{F} contains the character \texttt{"i"} and \texttt{false} otherwise.

The \texttt{multiline} property of the newly constructed object is set to a Boolean value that is \texttt{true} if \texttt{F} contains the character \texttt{"m"} and \texttt{false} otherwise.

If \texttt{F} contains any character other than \texttt{"g"}, \texttt{"i"}, or \texttt{"m"}, or if it contains the same one more than once, then throw a \texttt{SyntaxError} exception.

If \texttt{P}'s characters do not have the form \texttt{Pattern}, then throw a \texttt{SyntaxError} exception. Otherwise let the newly constructed object have a [[Match]] property obtained by evaluating ("compiling") \texttt{Pattern}. Note that evaluating \texttt{Pattern} may throw a \texttt{SyntaxError} exception. (Note: if \texttt{pattern} is a \texttt{StringLiteral}, the usual escape sequence substitutions are performed before the string is processed by \texttt{RegExp}. If \texttt{pattern} must contain an escape sequence to be recognised by \texttt{RegExp}, the \texttt{\"\"} character must be escaped within the \texttt{StringLiteral} to prevent its being removed when the contents of the \texttt{StringLiteral} are formed.)

The \texttt{source} property of the newly constructed object is set to an implementation-defined string value in the form of a \texttt{Pattern} based on \texttt{P}.

The \texttt{lastIndex} property of the newly constructed object is set to \texttt{0}.

15 December 2008
The [[Prototype]] property of the newly constructed object is set to the original RegExp prototype object, the one that is the initial value of `RegExp.prototype`.

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

15.10.5 Properties of the RegExp Constructor

The value of the internal [[Prototype]] property of the RegExp constructor is the Function prototype object (15.3.4).

Besides the internal properties and the length property (whose value is 2), the RegExp constructor has the following properties:

15.10.5.1 `RegExp.prototype`

The initial value of `RegExp.prototype` is the RegExp prototype object (15.10.6).

This property shall have the attributes `{ [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }`.

15.10.6 Properties of the RegExp Prototype Object

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

The RegExp prototype object is itself a regular expression object; its [[Class]] is "RegExp".

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

In the following descriptions of functions that are properties of the RegExp prototype object, the phrase "this RegExp object" refers to the object that is the this value for the invocation of the function; a TypeError exception is thrown if the this value is not an object for which the value of the internal [[Class]] property is "RegExp".

15.10.6.1 `RegExp.prototype.constructor`

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

15.10.6.2 `RegExp.prototype.exec(string)`

Performs a regular expression match of `string` against the regular expression and returns an Array object containing the results of the match, or null if the string did not match.

The string `ToString(string)` is searched for an occurrence of the regular expression pattern as follows:

1. Let `S` be the value of `ToString(string)`.
2. Let `length` be the length of `S`.
3. Let `lastIndex` be the value of the `lastIndex` property.
4. Let `i` be the value of `ToInteger(lastIndex)`.
5. If the `global` property is false, let `i = 0`.
6. If `i < 0` or `i > length` then set `lastIndex` to 0 and return null.
7. Call `[[Match]]`, giving it the arguments `S` and `i`. If `[[Match]]` returned failure, go to step 8; otherwise let `r` be its State result and go to step 10.
8. Let `i = i + 1`.
10. Let `e` be `r`'s endIndex value.
11. If the `global` property is true, set `lastIndex` to `e`.
12. Let `n` be the length of `r`'s captures array. (This is the same value as 15.10.2.1's NCapturingParents.)
13. Return a new array with the following properties:

   The `index` property is set to the position of the matched substring within the complete string `S`.
   The input property is set to `S`.
   The length property is set to `n + 1`.
   The 0 property is set to the matched substring (i.e. the portion of `S` between offset `i` inclusive and offset `e` exclusive).

For each integer `i` such that `I > 0` and `I ≤ n`, set the property named `ToString(i)` to the `i`th element of `r`'s captures array.
15.10.6.3 RegExp.prototype.test(string)
   Equivalent to the expression RegExp.prototype.exec(string) \(!=\) null.

15.10.6.4 RegExp.prototype.toString()
   Let src be a string in the form of a Pattern representing the current regular expression. src may or may
   not be identical to the source property or to the source code supplied to the RegExp constructor; however, if src were supplied to the RegExp constructor along with the current regular expression's flags, the resulting regular expression must behave identically to the current regular expression.

   toString returns a string value formed by concatenating the strings "/", src, and "/"; plus "/g" if the
   global property is true, "/i" if the ignoreCase property is true, and "/m" if the multiline
   property is true.

   NOTE
   An implementation may choose to take advantage of src being allowed to be different from the source
   passed to the RegExp constructor to escape special characters in src. For example, in the regular
   expression obtained from new RegExp("/"), src could be, among other possibilities, "/" or
   "\//". The latter would permit the entire result ("/\//") of the toString call to have the form
   RegularExpressionLiteral.

15.10.7 Properties of RegExp Instances
   RegExp instances inherit properties from their [[Prototype]] object as specified above and also have the
   following properties.

15.10.7.1 source
   The value of the source property is string in the form of a Pattern representing the current regular
   expression. This property shall have the attributes {
   [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}.

15.10.7.2 global
   The value of the global property is a Boolean value indicating whether the flags contained the
   character "/g". This property shall have the attributes {
   [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}.

15.10.7.3 ignoreCase
   The value of the ignoreCase property is a Boolean value indicating whether the flags contained the
   character "/i". This property shall have the attributes {
   [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}.

15.10.7.4 multiline
   The value of the multiline property is a Boolean value indicating whether the flags contained the
   character "/m". This property shall have the attributes {
   [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false}.

15.10.7.5 lastIndex
   The value of the lastIndex property is an integer that specifies the string position at which to start
   the next match. This property shall have the attributes {
   [[Writable]]: true, [[Enumerable]]: false, [[Configurable]]: false}.

15.11 Error Objects
   Instances of Error objects are thrown as exceptions when runtime errors occur. The Error objects may also
   serve as base objects for user-defined exception classes.

15.11.1 The Error Constructor Called as a Function
   When Error is called as a function rather than as a constructor, it creates and initialises a new Error
   object. Thus the function call Error(...) is equivalent to the object creation expression new
   Error(...) with the same arguments.
15.11.1 Error (message)

The [[Prototype]] property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of Error.prototype (15.11.3.1).

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

The [[Extensible]] property of the newly constructed object is set to true.

If the argument message is not undefined, the message property of the newly constructed object is set to ToString(message). Otherwise, the message property is set to the empty string.

15.11.2 The Error Constructor

When Error is called as part of a new expression, it is a constructor: it initialises the newly created object.

15.11.2.1 new Error (message)

The [[Prototype]] property of the newly constructed object is set to the original Error prototype object, the one that is the initial value of Error.prototype (15.11.3.1).

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

The [[Extensible]] property of the newly constructed object is set to true.

If the argument message is not undefined, the message property of the newly constructed object is set to ToString(message).

15.11.3 Properties of the Error Constructor

The value of the internal [[Prototype]] property of the Error constructor is the Function prototype object (15.3.4).

Besides the internal properties and the length property (whose value is 1), the Error constructor has the following property:

15.11.3.1 Error.prototype

The initial value of Error.prototype is the Error prototype object (15.11.4).

This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

15.11.4 Properties of the Error Prototype Object

The Error prototype object is itself an Error object (its [[Class]] is "Error").

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

15.11.4.1 Error.prototype.constructor

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

15.11.4.2 Error.prototype.name

The initial value of Error.prototype.name is "Error".

15.11.4.3 Error.prototype.message

The initial value of Error.prototype.message is an implementation-defined string.

15.11.4.4 Error.prototype.toString ( )

Returns an implementation-defined string.

15.11.5 Properties of Error Instances

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

15.11.6 Native Error Types Used in This Standard

One of the NativeError objects below is thrown when a runtime error is detected. All of these objects share the same structure, as described in 15.11.7.

15 December 2008
15.11.6.1 EvalError
Indicates that the global function eval was used in a way that is incompatible with its definition. See 15.1.2.1.

15.11.6.2 RangeError
Indicates a numeric value has exceeded the allowable range. See 15.4.2.2, 15.4.5.1, 15.7.4.5, 15.7.4.6, and 15.7.4.7.

15.11.6.3 ReferenceError
Indicate that an invalid reference value has been detected. See 8.7.1, and 8.7.2.

15.11.6.4 SyntaxError
Indicates that a parsing error has occurred. See 15.1.2.1, 15.3.2.1, 15.10.2.5, 15.10.2.9, 15.10.2.15, 15.10.2.19, and 15.10.4.1.

15.11.6.5 TypeError
Indicates the actual type of an operand is different than the expected type. See 8.6.2, 8.6.2.6, 9.9, 11.2.2, 11.2.3, 11.8.6, 11.8.7, 15.3.4.2, 15.3.4.3, 15.3.4.4, 15.3.5.3, 15.4.4.2, 15.4.4.3, 15.5.4.2, 15.5.4.3, 15.6.4, 15.6.4.2, 15.6.4.3, 15.7.4, 15.7.4.2, 15.7.4.4, 15.9.5, 15.9.5.9, 15.9.5.27, 15.10.4.1, and 15.10.6.

15.11.6.6 URIError
Indicates that one of the global URI handling functions was used in a way that is incompatible with its definition. See 15.1.3.

15.11.7 NativeError Object Structure
When an ECMAScript implementation detects a runtime error, it throws an instance of one of the NativeError objects defined in 15.11.6. Each of these objects has the structure described below, differing only in the name used as the constructor name instead of NativeError, in the name property of the prototype object, and in the implementation-defined message property of the prototype object.

For each error object, references to NativeError in the definition should be replaced with the appropriate error object name from 15.11.6.

15.11.7.1 NativeError Constructors Called as Functions
When a NativeError constructor is called as a function rather than as a constructor, it creates and initialises a new object. A call of the object as a function is equivalent to calling it as a constructor with the same arguments.

15.11.7.2 NativeError (message)
The [[Prototype]] property of the newly constructed object is set to the prototype object for this error constructor. The [[Class]] property of the newly constructed object is set to "Error". The [[Extensible]] property of the newly constructed object is set to true.

If the argument message is not undefined, the message property of the newly constructed object is set to ToString(message).

15.11.7.3 The NativeError Constructors
When a NativeError constructor is called as part of a new expression, it is a constructor: it initialises the newly created object.

15.11.7.4 New NativeError (message)
The [[Prototype]] property of the newly constructed object is set to the prototype object for this NativeError constructor. The [[Class]] property of the newly constructed object is set to "Error".

The [[Extensible]] property of the newly constructed object is set to true.

If the argument message is not undefined, the message property of the newly constructed object is set to ToString(message).

15.11.7.5 Properties of the NativeError Constructors
The value of the internal [[Prototype]] property of a NativeError constructor is the Function prototype object (15.3.4).

15 December 2008
Besides the internal properties and the `length` property (whose value is 1), each `NativeError` constructor has the following property:

**15.11.7.6** NativeError.prototype

The initial value of `NativeError.prototype` is a `NativeError` prototype object (15.11.7.7). Each `NativeError` constructor has a separate prototype object. This property has the attributes { `[[Writable]]`: false, `[[Enumerable]]`: false, `[[Configurable]]`: false }.

**15.11.7.7** Properties of the NativeError Prototype Objects

Each `NativeError` prototype object is an `Error` object (its `[[Class]]` is "Error"). The value of the internal `[[Prototype]]` property of each `NativeError` prototype object is the `Error` prototype object (15.11.4).

**15.11.7.8** NativeError.prototype.constructor

The initial value of the `constructor` property of the prototype for a given `NativeError` constructor is the `NativeError` constructor function itself (15.11.7).

**15.11.7.9** NativeError.prototype.name

The initial value of the `name` property of the prototype for a given `NativeError` constructor is the name of the constructor (the name used instead of `NativeError`).

**15.11.7.10** NativeError.prototype.message

The initial value of the `message` property of the prototype for a given `NativeError` constructor is an implementation-defined string.

**NOTE**

The prototypes for the `NativeError` constructors do not themselves provide a `toString` function, but instances of errors will inherit it from the `Error` prototype object.

**15.11.7.11** Properties of NativeError Instances

`NativeError` instances have no special properties beyond those inherited from the `Error` prototype object.

### 15.12 The JSON Object

The JSON object is a single object that contains two functions, `parse` and `stringify`, that are used to parse and construct JSON texts. The JSON Data Interchange Format is described in RFC 4627 [http://www.ietf.org/rfc/rfc4627.txt?number=4627].

The value of the internal `[[Prototype]]` property of the JSON object is the `Object` prototype object (15.2.3.1). The value of the internal `[[Class]]` property of the JSON object is "JSON". The value of the `[[Extensible]]` property of the JSON object is set to `true`.

The JSON object does not have a `[[Construct]]` property; it is not possible to use the JSON object as a constructor with the `new` operator.

The JSON object does not have a `[[Call]]` property; it is not possible to invoke the JSON object as a function.

JSON.stringify produces a string that conforms to the following grammar. JSON.parse accepts a string that conforms to the following grammar. JSON recognizes `<SP>`, `<TAB>`, `<CR>`, and `<LF>` as white space. Only those characters are recognized as white space.
The `parse` function parses a JSON text (a JSON formatted string) and produces a ECMAScript value.

The JSON format is a restricted form of ECMAScript literal. JSON objects are realized as ECMAScript objects. JSON Arrays are realized as ECMAScript arrays. JSON strings, numbers, booleans, and null are realized as ECMAScript strings, numbers, booleans, and null. JSON uses a more limited set of white
space characters than WhiteSpace. The process of parsing is similar to 11.1.4 and 11.1.5 as constrained by the JSON grammar.

The optional reviver parameter is a function that takes two parameters, (key, value). It can filter and transform the results. It is called with each of the key/value pairs produced by the parse, and its return value is used instead of the original value. If it returns what it received, the structure is not modified. If it returns undefined then the member is deleted from the result.

1. ToString of text.
2. Parse Result(1) as a JSON value. Throw a SyntaxError exception if the text did not conform to the JSON grammar for JSON values. JSON objects will produce objects as if made by the original Object constructor. JSON arrays will produce arrays as if made by the original Array constructor. JSON numbers will produce number values. JSON strings will produce string values. JSON true, false, and null will produce the true, false, and null values. No other values are possible.
3. If IsCallable(reviver) is true
   a. Produce a new object as if by the original Object constructor.
   b. Call the [[Put]] method of Result(3.a) with an empty string and Result(2).
   c. Call the abstract operation walk, passing Result(3.b) and the empty string. The abstract operation walk is described below.
   d. Return Result(3.c).
4. Else
   a. Return Result(2).

The abstract operation walk is a recursive abstract operation that takes two parameters: a holder object and the name of a property in that object.

1. Call the [[Get]] method of the holder with name.
2. If Result(1) is an object, and IsCallable(Result(1)) is false, then
   a. If the [[Class]] property of Result(1) is 'Array'
      i. Set I to 0.
      ii. While I is less than then length of Result(1)
         1. Call the internal walk function, passing Result(1) and I.
         2. Call the [[Put]] method of Result(1) with Result(2.a.ii.1).
         3. Add 1 to I.
   b. Else
      i. Produce an array using the original Object.keys method with Result(1).
      ii. For each element in Result(2.b.i)
         1. Call the internal walk function, passing Result(1) and Result(2.b.ii).
      2. If Result(2.b.ii) is undefined
         a. Call the [[Delete]] method of Result(1) with Result(2.b.ii).
      3. Else
         a. Call the [[Put]] method of Result(1) with Result(2.b.ii.1).
3. Call reviver as a method of holder with name and Result(1).
4. Return Result(3).

NOTE: In the case where there are duplicate name strings within an object, lexically preceeding values for the same key shall be overwritten.

15.12.2 stringify ( value [ , replacer [ , space ] ] )

The stringify function produces a JSON formatted string that captures information from a JavaScript value. It can take three parameters. The first parameter is required. The value parameter is a JavaScript value is usually an object or array, although it can also be a string, boolean, number or null. The optional replacer parameter is either a function that alters the way objects and arrays are stringified, or an array of strings that acts as a whitelist for selecting the keys that will be stringified. The optional space parameter is a string or number that allows the result to have white space injected into it to improve human readability.

JSON structures are allowed to be nested to any depth, but they must be acyclic. If the value is a cyclic structure, then the stringify function must throw an Error. This is an example of a value that cannot be stringified:

```javascript
a = [ ];
```
a[0] = a;
my_text = JSON.stringify(a); // This must throw an Error.

The null value is rendered in JSON text as the string null.
The true value is rendered in JSON text as the string true.
The false value is rendered in JSON text as the string false.

String values are wrapped in double quotes. The characters " and \ are escaped with \ prefixes. Control characters are replaced with escape sequences \\uHHHH, or with the shorter forms, \b (backspace), \t (formfeed), \n (newline), \r (carriage return), \t (tab).

Finite numbers are stringifyed by String(number). NaN and Infinity regardless of sign are represented as the string null.

Values that do not have a JSON representation (such as undefined and functions) do not produce a string. Instead they produce the undefined value. In arrays these values are represented as the string null. In objects an unrepresentable value causes the property to be excluded from stringification.

An object is rendered as an opening left brace followed by zero or more properties, separated with commas, closed with a right bracket. A property is a quoted string representing the key or property name, a colon, and then the stringified property value. An array is rendered as an opening left bracket followed by zero or more values, separated with commas, closed with a right bracket.

These are the steps in stringifying an object:
1. Create a new array by the original Array method.
2. Let stack be Result(1).
3. Let indent be the empty string.
4. If space is a number
   a. Set gap to a string containing space space characters. This will be the empty string if space is less than 1.
5. Else if space is a string
   a. Set gap to space.
6. Else
   a. Set gap to the empty string.
7. Create a new Object by the original Object method.
8. Call the [[Put]] method of Result(7) with the empty string and value.
9. Call the abstract operation str with the empty string and Result(7).
10. Return Result(9).

The internal abstract operation str(key, holder) has access to the replacer from the invocation of the stringify method. Its algorithm is as follows:
1. Call the [[Get]] method of holder with key.
2. Let value be Result(1).
3. If value is an object
   a. Call the [[Get]] method on value with "toJSON".
   b. If IsCallable(Result(3.a)) is true
      i. Call Result(3.a) as a method of value with key.
      ii. Let value be Result(3.b).
4. If IsCallable(replacer) is true
   a. Call replacer as a method of holder with key and value.
   b. Let value be Result(4.a).
5. If value is null then return "null".
6. If value is true then return "true".
7. If value is false then return "false".
8. If value is a string, then return the result of calling the abstract operation Quote with value.
9. If value is a number
    a. If value is finite then return value.
    b. Return "null".
10. If value is an object, and IsCallable(value) is false
    a. If the [[Class]] property of value is "Array" then
i. Call the abstract operation JA with value.
   ii. Return Result(10.a.1).
   b. Call the abstract operation JO with value.
   c. Return Result(10.b).
11. Return undefined.

The abstract operation Quote(value) wraps a string value in double quotes and escapes characters within it.
1. Let product be the double quote character.
2. For each character in value
   a. If Result(2) is the double quote character or backslash character
      i. Let product be the concatenation of product and the backslash character.
      ii. Let product be the concatenation of product and Result(2).
   b. Else If Result(2) is backspace, formfeed, newline, carriage return, or tab
      i. Let product be the concatenation of product and the backslash character.
      ii. Let product be the concatenation of product and the lowercase letter b, f, n, r, or t.
   c. Else If Result(2) is a control character having a value less than the space character
      i. Let product be the concatenation of product and the backslash character.
      ii. Let product be the concatenation of product and the lowercase u character.
      iii. Convert the numeric value of Result(2) to a string of 4 base 16 digits.
      iv. Let product be the concatenation of product and Result(2.c.3).
   d. Else
      i. Let product be the concatenation of product and Result(2).
3. Let product be the concatenation of product and the double quote character.
4. Return product.

The abstract operation JO(value) serializes an object. It has access to the stack, indent, gap, replacer, and space of the invocation of the stringify method.
1. Call the original indexOf method on stack with value.
2. If Result(1) is not -1 then throw an Error because the structure is cyclical.
3. Push value onto stack.
4. Let stepback be indent.
5. Let indent be the concatenation of indent and gap.
6. If the [[Class]] property of replacer is "Array"
   a. Let K be the replacer parameter.
   b. Else
      a. Call the original Object.keys method with value.
      b. Let K be Result(6.a).
   8. Create a new array by the original Array method.
   9. Let partial be Result(8).
10. For each element of K.
    a. Call the str function with Result(10) and value.
    b. If Result(10.a) is not undefined
       i. Call Quote with Result(10).
       ii. Let member be Result(10.b.i).
       iii. Let member be the concatenation of member and the colon character.
       iv. If gap is not empty string
          1. Let member be the concatenation of member and the space character.
       v. Else
          1. Let member be the concatenation of member and the Result(10.a).
         vi. Push Result(10.b.v) onto partial.
11. If Length(partial) is 0 then
    a. Let final be "{".
12. Else
    a. If gap is the empty string
       i. Call the original join method of partial with the comma character.
       ii. Concatenate "[" and Result(12.a.i) and "]".
       iii. Set final to Result(12.a.ii).
    b. Else
....
i. Concatenate the comma character and the line feed character and indent.

ii. Call the original join method of partial with Result(12.b.i).

iii. Concatenate "[" and the line feed character and indent and Result(12.b.ii) and the line feed character and stepback and "]".

iv. Set final to Result(12.b.iii).

13. Pop the stack.
14. Let indent be stepback.
15. Return final.

The abstract operation JA(value) serializes an array. It has access to the stack, indent, gap, and space of the invocation of the stringify method. The representation of arrays includes only the elements between zero and array.length – 1. Named properties are excluded from the stringification. An array is stringified as an open left bracket, elements separated by comma, and a closing right bracket.

1. Call the original indexOf method on stack with value.
2. If Result(1) is not -1 then throw an Error because the structure is cyclical.
3. Push value onto stack.
4. Let stepback be indent.
5. Let indent be the concatenation of indent and gap.
6. Create a new array by the original Array method.
7. Let partial be Result(6).
8. For each index in value.
   a. Call the str function with Result(8) and value.
      i. Push null on partial.
   b. If Result(8.a) is undefined
      i. Push null on partial.
   c. Else
      i. Push Result(8.a).
9. If Length(partial) is 0 then
   a. Let final be "["].
10. Else
    a. If gap is the empty string
       i. Call the original join method of partial with the comma character.
       ii. Concatenate "[" and Result(12.a.i) and "]".
       iii. Set final to Result(12.a.ii).
    b. Else
       i. Concatenate the comma character and the line feed character and indent.
       ii. Call the original join method of partial with Result(10.b.i).
       iii. Concatenate "[" and the line feed character and indent and Result(10.b.ii) and the line feed character and stepback and "]".
       iv. Set final to Result(10.b.iii).
11. Pop the stack.
12. Let indent be stepback.
16 Errors

An implementation should report runtime errors at the time the relevant language construct is evaluated. An implementation may report syntax errors in the program at the time the program is read in, or it may, at its option, defer reporting syntax errors until the relevant statement is reached. An implementation may report syntax errors in eval code at the time eval is called, or it may, at its option, defer reporting syntax errors until the relevant statement is reached.

An implementation must treat any instance of the following kinds of runtime errors as a syntax error and therefore report it early:

- Attempts to define an ObjectLiteral that has multiple get property assignments with the same name or multiple set property assignments with the same name.
- Attempts to define an ObjectLiteral that has both an accessor property assignment and a get or set property assignment with the same name.
- Errors in regular expression literals.
- Violation of strict mode restriction whose detection does not require program execution.

An implementation may treat any instance of the following kinds of runtime errors as a syntax error and therefore report it early:

- Improper uses of return, break, and continue.
- Using the eval property other than via a direct call.
- Errors in regular expression literals.
- Attempts to call PutValue on a value that is not a reference (for example, executing the assignment statement 3=4).

An implementation shall not report other kinds of runtime errors early even if the compiler can prove that a construct cannot execute without error under any circumstances. An implementation may issue an early warning in such a case, but it should not report the error until the relevant construct is actually executed.

An implementation shall report all errors as specified, except for the following:

- An implementation may extend program and regular expression syntax. To permit this, all operations (such as calling eval, using a regular expression literal, or using the Function or RegExp constructor) that are allowed to throw SyntaxError are permitted to exhibit implementation-defined behaviour instead of throwing SyntaxError when they encounter an implementation-defined extension to the program or regular expression syntax.
- An implementation may provide additional types, values, objects, properties, and functions beyond those described in this specification. This may cause constructs (such as looking up a variable in the global scope) to have implementation-defined behaviour instead of throwing an error (such as ReferenceError).
- An implementation is not required to detect EvalError. If it chooses not to detect EvalError, the implementation must allow eval to be used indirectly and/or allow assignments to eval.
- An implementation may define behaviour other than throwing RangeError for toFixed, toExponential, and toPrecision when the fractionDigits or precision argument is outside the specified range.
Annex A
(informative)

Grammar Summary

A.1 Lexical Grammar

SourceCharacter ::= any Unicode character

InputElementDiv ::= See clause 6
WhiteSpace
LineTerminator
Comment
Token
DivPunctuator

InputElementRegExp ::= See clause 7
WhiteSpace
LineTerminator
Comment
Token
RegularExpressionLiteral

WhiteSpace ::= See 7.2
<TAB>
<VT>
<FF>
<SP>
<NEL>
<NBSP>
<ZWSP>
<BOM>
<USP>

LineTerminator ::= See 7.3
<LF>
<CR>
<LS>
<PS>

LineTerminatorSequence ::= See 7.3
<LF>
<CR> [lookahead \g<LF>] 
<LS>
<PS>
<CR> <LF>

Comment ::= See 7.4
MultiLineComment
SingleLineComment

15 December 2008
MultiLineComment ::
   /* MultiLineCommentChars */

MultiLineCommentChars ::
   MultiLineNotAsteriskChar MultiLineCommentChars opt
   * PostAsteriskCommentChars opt

PostAsteriskCommentChars ::
   MultiLineNotForwardSlashOrAsteriskChar MultiLineCommentChars opt
   * PostAsteriskCommentChars opt

MultiLineNotAsteriskChar ::
   SourceCharacter but not asterisk *

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

SingleLineComment ::
   // SingleLineCommentChars

SingleLineCommentChars ::
   SingleLineCommentChar SingleLineCommentChars opt

SingleLineCommentChar ::
   SourceCharacter but not LineTerminator

Token ::
   IdentifierName
   Punctuator
   NumericLiteral
   StringLiteral

ReservedWord ::
   Keyword
   FutureReservedWord
   NullLiteral
   BooleanLiteral

Keyword :: one of
   break   else   new   var
   case    finally return void
   catch   for    switch while
   continue function this with
   default if    throw debugger
   do      in     try
   const  enum    int    short

FutureReservedWord :: one of
   abstract enum    int    short

15 December 2008
boolean  export  interface  static
byte    extends  long    super
char    final    native  synchronized
class   float    package  throws
cost    goto     private transient
double  implements  protected volatile

delimited: debugger

Identifier :: See 7.6
  IdentifierName but not ReservedWord

IdentifierName :: See 7.6
  IdentifierStart
  IdentifierName IdentifierPart

IdentifierStart :: See 7.6
  UnicodeLetter
    $
  \
  UnicodeEscapeSequence

IdentifierPart :: See 7.6
  IdentifierStart
  UnicodeCombiningMark
  UnicodeDigit
  UnicodeConnectorPunctuation
  UnicodeEscapeSequence

UnicodeLetter See 7.6
  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 See 7.6
  any character in the Unicode categories “Non-spacing mark (Mn)” or “Combining spacing mark (Mc)”

UnicodeDigit See 7.6
  any character in the Unicode category “Decimal number (Nd)”

UnicodeConnectorPunctuation See 7.6
  any character in the Unicode category “Connector punctuation (Pc)”

UnicodeEscapeSequence :: See 7.6
\u HexDigit HexDigit HexDigit 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

Punctuator :: one of See 7.7
  { ) ( [ ]
  . ; , < > <=
  >= == != === !==
  + - * % ++ --

15 December 2008
DivPunctuator :: one of
    / /=

Literal ::
    NullLiteral
    BooleanLiteral
    NumericLiteral
    StringLiteral

NullLiteral ::
    null

BooleanLiteral ::
    true
    false

NumericLiteral ::
    DecimalLiteral
    HexIntegerLiteral

DecimalLiteral ::
    DecimalIntegerLiteral . DecimalDigits opt ExponentPart opt
    . DecimalDigits ExponentPart opt
    DecimalIntegerLiteral ExponentPart opt

DecimalIntegerLiteral ::
    0
    NonZeroDigit DecimalDigits

DecimalDigits ::
    DecimalDigit
    DecimalDigits DecimalDigit

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

ExponentIndicator :: one of
    e E

See 7.7

See 7.8

See 7.8.1

See 7.8.2

See 7.8.3

See 7.8.3
SignedInteger ::
  DecimalDigits
  + DecimalDigits
  - DecimalDigits

HexIntegerLiteral ::
  0x HexDigit
  0X HexDigit
  HexIntegerLiteral HexDigit

StringLiteral ::
  " DoubleStringCharacters opt "
  ' SingleStringCharacters opt '

DoubleStringCharacters ::
  DoubleStringCharacter DoubleStringCharacters opt

SingleStringCharacters ::
  SingleStringCharacter SingleStringCharacters opt

DoubleStringCharacter ::
  SourceCharacter but not double-quote " or backslash \ or LineTerminator
  \ EscapeSequence
  LineContinuation

SingleStringCharacter ::
  SourceCharacter but not single-quote ' or backslash \ 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

15 December 2008
NonEscapeCharacter ::
  SourceCharacter but not EscapeCharacter or LineTerminator

EscapeCharacter ::
  SingleEscapeCharacter
  DecimalDigit
    x
    u

HexEscapeSequence ::
  x HexDigit HexDigit

UnicodeEscapeSequence ::
  u HexDigit HexDigit HexDigit HexDigit

RegularExpressionLiteral ::
  / RegularExpressionBody / RegularExpressionFlags

RegularExpressionBody ::
  RegularExpressionFirstChar RegularExpressionChars

RegularExpressionChars ::
  [empty]
  RegularExpressionChars RegularExpressionChar

RegularExpressionFirstChar ::
  NonTerminator but not * or \ or / or [
    BackslashSequence
    RegularExpressionClass

RegularExpressionChar ::
  NonTerminator but not \ or / or [
    BackslashSequence
    RegularExpressionClass

BackslashSequence ::
  \ NonTerminator

NonTerminator ::
  SourceCharacter but not LineTerminator

RegularExpressionClass ::
  [ RegularExpressionClassPreamble RegularExpressionClassChars ]

RegularExpressionClassPreamble ::
  [empty]
  *
  *
  ^

15 December 2008
RegularExpressionClassChars ::
[empty]
  RegularExpressionClassChars RegularExpressionClassChar

RegularExpressionClassChar ::
  NonTerminator but not | or | or -
  - RegularExpressionClassChar
  BackslashExpression

RegularExpressionFlags ::
[empty]
  RegularExpressionFlags IdentifierPart

A.2 Number Conversions
StringNumericLiteral :::
  See 9.3.1
  StrWhiteSpace opt
  StrDecimalLiteral StrUnsignedDecimalLiteral Opt
  StrUnsignedDecimalLiteral

StrWhiteSpace :::
  See 9.3.1
  StrWhiteSpaceChar StrWhiteSpace opt

StrWhiteSpaceChar :::
  See 9.3.1
  StrWhiteSpace StrWhiteSpace opt

StrNumericLiteral :::
  See 9.3.1
  StrDecimalLiteral
  HexIntegerLiteral

StrDecimalLiteral :::
  See 9.3.1
  StrUnsignedDecimalLiteral
  + StrUnsignedDecimalLiteral
  - StrUnsignedDecimalLiteral

StrUnsignedDecimalLiteral :::
  See 9.3.1
  Infinity
  DecimalDigits , DecimalDigits Opt ExponentPart opt
  . DecimalDigits ExponentPart opt
  DecimalDigits ExponentPart opt

DecimalDigits :::
  See 9.3.1
  DecimalDigit
  DecimalDigit DecimalDigit

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

ExponentPart :::
  See 9.3.1
  ExponentIndicator SignedInteger

15 December 2008
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

A.3 Expressions

PrimaryExpression :
   this
   Identifier
   Literal
   ArrayLiteral
   ObjectLiteral
   { Expression }

ArrayLiteral :
   [ Elision?, ]
   [ ElementList ]
   [ ElementList , Elision?, ]

ElementList :
   Elision?, AssignmentExpression
   ElementList , Elision?, AssignmentExpression

Elision :
   ,
   Elision ,

ObjectLiteral :
   { }
   ( propertyNameAndValueList )
   ( propertyNameAndValueList , )

propertyNameAndValueList :
   PropertyAssignment
   PropertyNameAndValueList , PropertyAssignment
Proper
ty Assignment:

See 11.1.5

PropertyName : AssignmentExpression

get PropertyName() { FunctionBody }

set PropertyName(PropertySetParameterList) { FunctionBody }

Property.Name :

See 11.1.5

Identifier
StringLiteral
NumericLiteral

Property.SetParameterList :

Identifier

See 11.1.5

MemberExpression :

See 11.2

PrimaryExpression
FunctionExpression
MemberExpression [ Expression ]
MemberExpression , Identifier
new MemberExpression Arguments

NewExpression :

MemberExpression
new NewExpression

CallExpression :

See 11.2

MemberExpression Arguments
CallExpression Arguments
CallExpression [ Expression ]
CallExpression , Identifier

Arguments :

See 11.2

() ( ArgumentList )

ArgumentList :

See 11.2

AssignmentExpression
ArgumentList , AssignmentExpression

LeftHandSideExpression :

See 11.2

NewExpression
CallExpression

PostfixExpression :

See 11.3

LeftHandSideExpression
LeftHandSideExpression [ no LineTerminator here] ++
LeftHandSideExpression [ no LineTerminator here] --
UnaryExpression : See 11.4

    PostfixExpression
    delete UnaryExpression
    void UnaryExpression
typeof UnaryExpression
++ UnaryExpression
-- UnaryExpression
+ UnaryExpression
- UnaryExpression
~ UnaryExpression
! UnaryExpression

MultiplicativeExpression : See 11.5

    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression

AdditiveExpression : See 11.6

    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression

ShiftExpression : See 11.7

    AdditiveExpression
    ShiftExpression << AdditiveExpression
    ShiftExpression >> AdditiveExpression
    ShiftExpression >>> AdditiveExpression

RelationalExpression : See 11.8

    ShiftExpression
    RelationalExpression < ShiftExpression
    RelationalExpression > ShiftExpression
    RelationalExpression <= ShiftExpression
    RelationalExpression >= ShiftExpression
    RelationalExpression instanceof ShiftExpression
    RelationalExpression in ShiftExpression

RelationalExpressionNoIn : See 11.8

    ShiftExpression
    RelationalExpressionNoIn < ShiftExpression
    RelationalExpressionNoIn > ShiftExpression
    RelationalExpressionNoIn <= ShiftExpression
    RelationalExpressionNoIn >= ShiftExpression
    RelationalExpressionNoIn instanceof ShiftExpression

EqualityExpression : See 11.9

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

15 December 2008
EqualityExpressionNoIn : See 11.9
    RelationalExpressionNoIn
    EqualityExpressionNoIn == RelationalExpressionNoIn
    EqualityExpressionNoIn != RelationalExpressionNoIn
    EqualityExpressionNoIn === RelationalExpressionNoIn
    EqualityExpressionNoIn !== RelationalExpressionNoIn

BitwiseANDEExpression : See 11.10
    EqualityExpression
    BitwiseANDEExpression & EqualityExpression

BitwiseANDEExpressionNoIn : See 11.10
    EqualityExpressionNoIn
    BitwiseANDEExpressionNoIn & EqualityExpressionNoIn

BitwiseXORExpression : See 11.10
    BitwiseANDEexpression
    BitwiseXORExpression ^ BitwiseANDEexpression

BitwiseXORExpressionNoIn : See 11.10
    BitwiseANDEexpressionNoIn
    BitwiseXORExpressionNoIn ^ BitwiseANDEexpressionNoIn

BitwiseORExpression : See 11.10
    BitwiseXORExpression
    BitwiseORExpression | BitwiseXORExpression

BitwiseORExpressionNoIn : See 11.10
    BitwiseXORExpressionNoIn
    BitwiseORExpressionNoIn | BitwiseXORExpressionNoIn

LogicalANDExpression : See 11.11
    BitwiseORExpression
    LogicalANDExpression && BitwiseORExpression

LogicalANDExpressionNoIn : See 11.11
    BitwiseORExpressionNoIn
    LogicalANDExpressionNoIn && BitwiseORExpressionNoIn

LogicalORExpression : See 11.11
    LogicalANDExpression
    LogicalORExpression | LogicalANDExpression

LogicalORExpressionNoIn : See 11.11
    LogicalANDExpressionNoIn
    LogicalORExpressionNoIn | LogicalANDExpressionNoIn
ConditionalExpression : See 11.12
    LogicalORExpression
    LogicalORExpression ? AssignmentExpression : AssignmentExpression

ConditionalExpressionNoIn : See 11.12
    LogicalORExpressionNoIn
    LogicalORExpressionNoIn ? AssignmentExpressionNoIn : AssignmentExpressionNoIn

AssignmentExpression : See 11.13
    ConditionalExpression
    LeftHandSideExpression AssignmentOperator AssignmentExpression

AssignmentExpressionNoIn : See 11.13
    ConditionalExpressionNoIn
    LeftHandSideExpression AssignmentOperator AssignmentExpressionNoIn

AssignmentOperator : one of See 11.13
    = /= %= += -= <<= >>= >>>= &= ^= |=

Expression : See 11.14
    AssignmentExpression
    Expression , AssignmentExpression

ExpressionNoIn : See 11.14
    AssignmentExpressionNoIn
    ExpressionNoIn , AssignmentExpressionNoIn

A.4 Statements
Statement : See clause 12
    Block
    VariableStatement
    EmptyStatement
    ExpressionStatement
    IfStatement
    IterationStatement
    ContinueStatement
    BreakStatement
    ReturnStatement
    WebStatement
    LabeledStatement
   SwitchStatement
    ThrowStatement
    TryStatement
    DebuggerStatement

Block : See 12.1
    { StatementListopt }

StatementList : See 12.1
    Statement
    StatementList Statement
VariableStatement : See 12.2
  var VariableDeclarationList ;

VariableDeclarationList : See 12.2
  VariableDeclaration
  VariableDeclarationList , VariableDeclaration

VariableDeclarationListNoIn : See 12.2
  VariableDeclarationNoIn
  VariableDeclarationListNoIn , VariableDeclarationNoIn

VariableDeclaration : See 12.2
  Identifier Initialiser opt

VariableDeclarationNoIn : See 12.2
  Identifier InitialiserNoIn opt

Initialiser : See 12.2
  = AssignmentExpression

InitialiserNoIn : See 12.2
  = AssignmentExpressionNoIn

EmptyStatement : See 12.3
  ;

ExpressionStatement : See 12.4
  [lookahead \{ , function\} Expression ;

IfStatement : See 12.5
  if ( Expression ) Statement else Statement
  if ( Expression ) Statement

IterationStatement : See 12.6
  do Statement while ( Expression ) ;
  while ( Expression ) Statement
  for ( ExpressionNoInopt ; Expressionopt ; Expressionopt ) Statement
  for ( var VariableDeclarationListNoIn ; Expressionopt ; Expressionopt ) Statement
  for ( LeftHandSideExpression in Expression ) Statement
  for ( var VariableDeclarationNoIn in Expression ) Statement

ContinueStatement : See 12.7
  continue [no LineTerminator here] Identifieropt ;

BreakStatement : See 12.8
  break [no LineTerminator here] Identifieropt ;

15 December 2008
ReturnStatement:  
\[ \text{return} \ [\text{no LineTerminator here}] \ \text{Expression}_{opt}; \]  
See 12.9

WithStatement:  
\[ \text{with} \ (\ \text{Expression}) \ \text{Statement} \]  
See 12.10

SwitchStatement:  
\[ \text{switch} \ (\ \text{Expression}) \ \text{CaseBlock} \]  
See 12.11

CaseBlock:  
\{ \ \text{CaseClauses}_{opt} \}  
\{ \ \text{CaseClauses}_{opt} \ \text{DefaultClause} \ \text{CaseClauses}_{opt} \} \]  
See 12.11

CaseClauses:  
\text{CaseClause} \ \text{CaseClauses} \ \text{CaseClause} \]  
See 12.11

CaseClause:  
\text{case} \ \text{Expression} : \ \text{StatementList}_{opt} \]  
See 12.11

DefaultClause:  
\text{default} : \ \text{StatementList}_{opt} \]  
See 12.11

LabelledStatement:  
\text{Identifier} : \ \text{Statement} \]  
See 12.12

ThrowStatement:  
\text{throw} \ [\text{no LineTerminator here}] \ \text{Expression} ; \]  
See 12.13

TryStatement:  
\text{try} \ \text{Block} \ \text{Catch} \ \text{try} \ \text{Block} \ \text{Finally} \ \text{try} \ \text{Block} \ \text{Catch} \ \text{Finally} \]  
See 12.14

Catch:  
\text{catch} \ (\ \text{Identifier}) \ \text{Block} \]  
See 12.14

Finally:  
\text{finally} \ \text{Block} \]  
See 12.14

DebuggerStatement:  
\text{debugger} ; \]  
See 12.15

A.5 Functions and Programs

FunctionDeclaration:  
\text{function} \ \text{Identifier} \ (\ \text{FormalParameterList}_{opt} \) \ \{ \ \text{FunctionBody} \} \]  
See clause 13
FunctionExpression : 
    function Identifier opt ( FormalParameterList opt ) { FunctionBody } See clause 13

FormalParameterList : 
    Identifier 
    FormalParameterList , Identifier See clause 13

FunctionBody : 
    UseStrictDirective opt SourceElements See clause 13

Program : 
    UseStrictDirective opt SourceElements See clause 14

UseStrictDirective : 
    " use strict useExtension opt " ; See clause 14

useExtension : 
    , DoubleStringCharacters opt

SourceElements : 
    SourceElement 
    SourceElements SourceElement

SourceElement : 
    Statement 
    FunctionDeclaration

A.6 Universal Resource Identifier Character Classes

uri :::=
    uriCharacters opt

uriCharacters :::=
    uriCharacter uriCharacters opt

uriCharacter :::=
    uriReserved 
    uriUnescaped 
    uriEscaped

uriReserved :::= one of 
    ; / ? @ & = + $ ,

uriUnescaped :::=
    uriAlpha
    DecimalDigit
    uriMark

uriEscaped :::=
    % HexDigit HexDigit

15 December 2008
A.7  Regular Expressions

Pattern ::=
\nDisjunction

Disjunction ::=
\nAlternative
Alternative | Disjunction

Alternative ::=
\n{empty}
Alternative Term

Term ::=
\nAssertion
Atom
Atom Quantifier

Assertion ::=
\n^ \$
\b \B

Quantifier ::=
\nQuantifierPrefix
QuantifierPrefix \?

QuantifierPrefix ::=
\n*
+
? ( DecimalDigits )
( DecimalDigits , )
( DecimalDigits , DecimalDigits )

Atom ::=
\nPatternCharacter
\ AtomEscape
CharacterClass
( Disjunction )
( ? : Disjunction )
( ? = Disjunction )
( ? ! Disjunction )
PatternCharacter :: SourceCharacter but not any of:  
\[ ^ \$ \, \. \, \* \, \+ \, \? \, ( \, ) \, [ \, ] \, { \, } \, | \]  
See 15.10.1

AtomEscape ::  
DecimalEscape  
CharacterEscape  
CharacterClassEscape  
See 15.10.1

CharacterEscape ::  
ControlEscape  
\( \circ \) ControlLetter  
HexEscapeSequence  
UnicodeEscapeSequence  
IdentityEscape  
See 15.10.1

ControlEscape :: one of  
\[ f \, n \, r \, t \, v \]  
See 15.10.1

ControlLetter :: one of  
\[ a \, b \, c \, d \, e \, f \, g \, h \, i \, j \, k \, l \, m \, n \, o \, p \, q \, r \, s \, t \, u \, v \, w \, x \, y \, z \]  
\[ A \, B \, C \, D \, E \, F \, G \, H \, I \, J \, K \, L \, M \, N \, O \, P \, Q \, R \, S \, T \, U \, V \, W \, X \, Y \, Z \]  
See 15.10.1

IdentityEscape ::  
SourceCharacter but not IdentifierPart  
See 15.10.1

DecimalEscape ::  
DecimalIntegerLiteral (lookahead @ DecimalDigit)  
See 15.10.1

CharacterClass ::  
\[ \{ \] (lookahead @ ClassRanges)  
\[ ^ \} \] ClassRanges  
\]  
See 15.10.1

ClassRanges ::  
\[ \{ \emptyset \} \] NonemptyClassRanges  
See 15.10.1

NonemptyClassRanges ::  
ClassAtom  
ClassAtom NonemptyClassRangesNoDash  
ClassAtom - ClassAtom ClassRanges  
See 15.10.1

NonemptyClassRangesNoDash ::  
ClassAtom  
ClassAtomNoDash NonemptyClassRangesNoDash  
ClassAtomNoDash - ClassAtom ClassRanges  
See 15.10.1

ClassAtom ::  
ClassAtomNoDash  
See 15.10.1

15 December 2008
ClassAtomNoDash ::
  SourceCharacter  \ but not one of \  ] ~
  \ ClassEscape

ClassEscape ::
  DecimalEscape
  \n  CharacterEscape
  CharacterClassEscape
Annex B
(informative)

Compatibility

B.1 Additional Syntax
Past editions of ECMAScript have included additional syntax and semantics for specifying octal literals and
octal escape sequences. These have been removed from this edition of ECMAScript. This non-normative
annex presents uniform syntax and semantics for octal literals and octal escape sequences for compatibility
with some older ECMAScript programs.

B.1.1 Numeric Literals
The syntax and semantics of 7.8.3 can be extended as follows:

Syntax

```
NumericLiteral ::
  DecimalLiteral
  HexIntegerLiteral
  OctalIntegerLiteral

OctalIntegerLiteral ::
  0 OctalDigit
  OctalIntegerLiteral OctalDigit

OctalDigit :: one of 0 1 2 3 4 5 6 7
```

Semantics

```
The MV of NumericLiteral :: OctalIntegerLiteral is the MV of OctalIntegerLiteral.
The MV of OctalDigit :: 0 is 0.
The MV of OctalDigit :: 1 is 1.
The MV of OctalDigit :: 2 is 2.
The MV of OctalDigit :: 3 is 3.
The MV of OctalDigit :: 4 is 4.
The MV of OctalDigit :: 5 is 5.
The MV of OctalDigit :: 6 is 6.
The MV of OctalDigit :: 7 is 7.
The MV of OctalIntegerLiteral :: 0 OctalDigit is the MV of OctalDigit.
The MV of OctalIntegerLiteral :: OctalIntegerLiteral OctalDigit is (the MV of OctalIntegerLiteral times 8) plus
the MV of OctalDigit.
```

B.1.2 String Literals
The syntax and semantics of 7.8.4 can be extended as follows:

Syntax

```
EscapeSequence ::
  CharacterEscapeSequence
  OctalEscapeSequence
  HexEscapeSequence
  UnicodeEscapeSequence
```
OctalEscapeSequence ::
    OctalDigit [lookahead ∈ DecimalDigit]
    ZeroToThree OctalDigit [lookahead ∈ DecimalDigit]
    FourToSeven OctalDigit
    ZeroToThree OctalDigit OctalDigit

ZeroToThree :: one of
    0 1 2 3

FourToSeven :: one of
    4 5 6 7

Semantics
The CV of EscapeSequence :: OctalEscapeSequence is the CV of the OctalEscapeSequence.
The CV of OctalEscapeSequence :: OctalDigit [lookahead ∈ DecimalDigit] is the character whose code point value is the MV of the OctalDigit.
The CV of OctalEscapeSequence :: ZeroToThree OctalDigit [lookahead ∈ DecimalDigit] is the character whose code point value is (8 times the MV of the ZeroToThree) plus the MV of the OctalDigit.
The CV of OctalEscapeSequence :: FourToSeven OctalDigit is the character whose code point value is (8 times the MV of the FourToSeven) plus the MV of the OctalDigit.
The CV of OctalEscapeSequence :: ZeroToThree OctalDigit OctalDigit is the character whose code point value is (64 (that is, 8²) times the MV of the ZeroToThree) plus (8 times the MV of the first OctalDigit) plus the MV of the second OctalDigit.
The MV of ZeroToThree :: 0 is 0.
The MV of ZeroToThree :: 1 is 1.
The MV of ZeroToThree :: 2 is 2.
The MV of ZeroToThree :: 3 is 3.
The MV of FourToSeven :: 4 is 4.
The MV of FourToSeven :: 5 is 5.
The MV of FourToSeven :: 6 is 6.
The MV of FourToSeven :: 7 is 7.

B.2 Additional Properties
Some implementations of ECMAScript have included additional properties for some of the standard native objects. This non-normative annex suggests uniform semantics for such properties without making the properties or their semantics part of this standard.

B.2.1 escape (string)
The escape function is a property of the global object. It computes a new version of a string value in which certain characters have been replaced by a hexadecimal escape sequence.

For those characters being replaced whose code point value is \0xFF or less, a two-digit escape sequence of the form %xx is used. For those characters being replaced whose code point value is greater than \0xFF, a four-digit escape sequence of the form \uxxxx is used.

When the escape function is called with one argument string, the following steps are taken:
1. Call ToString(string).
2. Compute the number of characters in Result(1).
3. Let R be the empty string.
4. Let k be 0.
5. If k equals Result(2), return R.
6. Get the character (represented as a 16-bit unsigned integer) at position k within Result(1).
7. If Result(6) is one of the 69 nonblank characters "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789@*_+-./" then go to step 13.

15 December 2008
8. If Result(6), is less than 256, go to step 11.
9. Let $S$ be a string containing six characters “%u%xyz” where $xyz$ are four hexadecimal digits encoding the value of Result(6).
11. Let $S$ be a string containing three characters “%xy” where $xy$ are two hexadecimal digits encoding the value of Result(6).
13. Let $S$ be a string containing the single character Result(6).
15. Let $S$ be a string containing the single character Result(6).
16. Go to step 5.

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

### B.2.2 unescape (string)

The `unescape` function is a property of the global object. It computes a new version of a string value in which each escape sequence of the sort that might be introduced by the `escape` function is replaced with the character that it represents.

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

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

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

The `substr` method takes two arguments, `start` and `length`, and returns a substring of the result of converting this object to a string, starting from character position `start` and running for `length` characters (or through the end of the string if `length` is `undefined`). If `start` is negative, it is treated as (sourceLength+start) where `sourceLength` is the length of the string. The result is a string value, not a String object. The following steps are taken:

1. Call `ToString`, giving it the `this` value as its argument.
2. Call `ToInteger(start)`.
3. If `length` is `undefined`, use `+Infinity`; otherwise call `ToInteger(length)`.
4. Compute the number of characters in `Result(1)`.

15 December 2008
5. If Result(2) is positive or zero, use Result(2); else use max(Result(4)+Result(2),0).
6. Compute min(max(Result(3),0), Result(4)–Result(5)).
7. If Result(6) ≤ 0, return the empty string "".
8. Return a string containing Result(6) consecutive characters from Result(1) beginning with the character
   at position Result(5).

The length property of the substr method is 2.

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

B.2.4 Date.prototype.getYear ( )

NOTE
The getFullYear method is preferred for nearly all purposes, because it avoids the “year 2000
problem.”

When the getYear method is called with no arguments the following steps are taken:
1. Let t be this time value.
2. If t is NaN, return NaN.
3. Return YearFromTime(LocalTime(t)) – 1900.

B.2.5 Date.prototype.setYear (year)

NOTE
The setFullYear method is preferred for nearly all purposes, because it avoids the “year 2000
problem.”

When the setYear method is called with one argument year the following steps are taken:
1. Let t be the result of LocalTime(this time value); but if this time value is NaN, let t be +0.
2. Call ToNumber(year).
3. If Result(2) is NaN, set the [[PrimitiveValue]] property of the this value to NaN and return NaN.
4. If Result(2) is not NaN and 0 ≤ ToInteger(Result(2)) ≤ 99 then Result(4) is ToInteger(Result(2))
   + 1900. Otherwise, Result(4) is Result(2).
5. Compute MakeDay(Result(4), MonthFromTime(t), DateFromTime(t)).
6. Compute UTC(MakeDate(Result(5), TimeWithinDay(t))).
7. Set the [[PrimitiveValue]] property of the this value to TimeClip(Result(6)).
8. Return the value of the [[PrimitiveValue]] property of the this value.

B.2.6 Date.prototype.toGMTString ( )

NOTE
The property toUTCString is preferred. The toGMTString property is provided principally for
compatibility with old code. It is recommended that the toUTCString property be used in new
ECMAScript code.

The Function object that is the initial value of Date.prototype.toGMTString is the same Function
object that is the initial value of Date.prototype.toUTCString.
Annex C

This annex needs to be updated according to the rest of the document.

The Strict variant of ECMAScript

C.1 The strict mode

C.1.1 Excluded Features

1. When defined within an execution context subset restricted to the strict subset, a function may not have two or more formal parameters that have the same name. An attempt to create such a function will fail, either statically, if expressed as a FunctionDeclaration or FunctionExpression, or dynamically by throwing a SyntaxError exception, if expressed in a call to the Function constructor (10.1.3.1).

2. For functions defined within an execution subset restricted to the strict subset, if an arguments object is created, a callee property is not created. The arguments object does not share properties with the activation object. Changing the value of an arguments object property does not change the value of the corresponding argument value and vice versa (10.1.8.1).

3. If either the execution context for the eval code or the execution context in which the eval operator was executed is subset restricted to the strict subset, the eval code cannot instantiate variables, functions, or constants in the lexical context of its eval operator. Instead, a new environment object is appended to the head of the calling context’s scope chain and that environment object is used for environment bindings instantiation with the eval code (10.2.2.1).

4. If this is evaluated within an execution context that is subset restricted to the strict subset, then the this value is not coerced.

5. When a postfix increment operator occurs within an execution context that is subset restricted to the strict subset, its LeftHandSide must not be a reference to a property with the attribute value {[[Writable]]: false} nor to a non-existent property of an object whose [[Extensible]] property has the value false. In these cases a TypeError exception is thrown (11.3.1.1).

6. The same restrictions as specified in 11.3.1.1 apply for the postfix decrement operator (11.3.2.1).

7. When a delete operator occurs within an execution context that is subset restricted to the strict subset, its UnaryExpression is further limited to being a MemberExpression. In addition, if the property to be deleted has the attribute {[[Configurable]]: false}, a TypeError exception is thrown (11.4.1.1).

8. The same restrictions as specified in 11.3.1.1 apply for the prefix increment operator (11.4.4.1).

9. The same restrictions as specified in 11.3.1.1 apply for the prefix decrement operator (11.4.5.1).

10. When a simple assignment occurs within an execution context that is subset restricted to the strict subset, its LeftHandSide must not evaluate to a Reference whose base is null. If it does a ReferenceError exception is thrown. The LeftHandSide also may not be a reference to a property with the attribute value {[[Writable]]: false} nor to a non-existent property of an object whose [[Extensible]] property has the value false. In these cases a TypeError exception is thrown (11.13.1.1).

11. A VariableStatement within an execution context that is subset restricted to the strict subset, may not occur as the StatementList of a Block. The occurrence of a VariableStatement in such a context must be treated as a syntax error (12.1.1).

12. In strict mode a Statement that is part of an IfStatement production may not be a VariableStatement nor may it be a LabelledStatement whose Statement production is a VariableStatement. The LabelledStatement restriction also applies if such a VariableStatement is preceeded by multiple labels (12.5.1).

13. A Statement that is an element of an IterationStatement production may not be a VariableStatement nor may it be a LabelledStatement whose Statement production is a VariableStatement. The LabelledStatement restriction also applies if such an VariableStatement is preceeded by multiple labels (12.6).

14. An execution context that is subset restricted to the strict subset, may not execute a WithStatement. The occurrence of a WithStatement in such a context should be treated as a syntax error (12.10.1).

15 December 2008
15. If an execution context that is subset restricted to the strict subset uses the value of the `eval` property in any way other than a direct call (that is, other than by the explicit use of its name as an `Identifier` which is the `MemberExpression in a CallExpression`), or if the `eval` property is assigned to, an `EvalError` exception is thrown (15.1.2.1.1).

16. If an execution context that is subset restricted to the strict subset uses the `Function.caller`, `Function.arguments`, or `arguments.caller` properties in any way, a `TypeError` exception should be thrown.

C.1.2 Additional Execution Exceptions
Annex D
(informative)

Note:
This annex needs to be updated according to the rest of the document.

Correction and Clarifications in Edition 3.1 with Possible Compatibility Impact
Through out: The meaning of phrases such as “as if by the expression new Array ()” are subject to misinterpretation. The specification text for all interal references and invocations of standard built-in objects and methods has been clarified by making it implicit that the intent is that the actual built-in object is to be used rather than the current dynamic value of the correspondingly name property.

11.8.2, 11.8.3, 11.8.5 While ECMAScript generally uses a left to right evaluation order the specification language for the > and <= operators resulted a partial right to left order. The specification has been corrected for these operators such that it now specifies a full left to right evaluation order. However, this change of order is potentially observable if user-defined valueOf or toString methods with side-effects are invoked during the evaluation process.

11.2.3 Edition 3.1 reverses the order of steps 2 and 3 of the algorithm. The original order as specified in Editions 1 through 3 was incorrectly specified such that side-effects of evaluating Arguments could effect the result of evaluating MemberExpression.

12.2 In Edition 3 the algorithm for evaluating the production VariableDeclaration : Identifier Initialiser was specified in a manner that is incorrect for situations where a VariableDeclaration is nested within a WithStatement for an object that has a property name that is identical to the Identifier in the VariableDeclaration. In this situation, the Edition 3 specification causes the value of the Initialiser to be assigned to the object’s property rather than the actual variable introduced by the declaration. For Edition 3.1 the algorithm has been revised such that the value of the Initialiser will be assigned to the associated variable regardless of any such nesting. The existing ECMAScript code that depends up faithful implementation of this Edition 3 semantics will not operate as expected using an implementation that conforms to the Edition 3.1 specification.

15.10.6 RegExp.prototype is now a RegExp object rather than an instance of Object. The value of its [[Class]] internal property which is observable using Object.prototype.toString is now “RegExp” rather than “Object”
Annex E
(informative)

Note:
This annex needs to be updated according to the rest of the document.

Section 7.1 Unicode format control characters are no longer stripped from ECMA-Script source text before processing.
Section 7.2 Unicode characters \(<\text{NEL}\rangle\), \(<\text{ZWSP}\rangle\), and \(<\text{BOM}\rangle\) are now treated as whitespace.
Section 7.3 Line terminator characters that are preceded by an escape sequence are now allowed within a string literal token.
Section 7.8.5 Regular expression literals now return an unique object each time the literal is evaluated. This change is detectable by any programs that test the object identity of such literal values or that are sensitive to the shared side effects.
Section 7.8.5 in ES3.1 requires scan time reporting of any possible RegExp constructor errors that would be produced when converting a RegularExpressionLiteral to a RegExp object. Prior to ES3.1 implementations were permitted to defer the reporting of such errors until the actual execution time creation of the object.
Section 12.6.4 for-in statements no longer throw a TypeError if the \(\text{in}\) expression evaluates to null or undefined. Instead, the statement behaves as if the value of the expression was an object with no enumerable properties.
Section 15: Implementations are now required to ignore extra arguments to standard built-in methods unless otherwise explicitly specified. In the 3rd Edition the handling of extra arguments were undefined and implementations were explicitly allowed to throw a TypeError exception.
Section 15.1.1: The value properties NaN, Infinity, and undefined of the Global Object have been changed to be read-only properties.
Section 15.10.2.12 \(\text{\$}\) now matches \(<\text{NEL}\rangle\), \(<\text{ZWSP}\rangle\), and \(<\text{BOM}\rangle\) as well as the two character sequence \(<\text{CR}\text{\langle LF\rangle}\) .
Section 15.9.4.2: Date parse is not required to first attempt to parse its argument as an ISO format string. Programs that use this format but depended upon implementation specific behavior (including failure) may behave differently.

Comment [pL26]: From AWB: What is the justification for this change?
Free printed copies can be ordered from:

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland
Fax: +41 22 849.60.01

Files of this Standard can be freely downloaded from our ECMA web site (www.ecma.ch). This site gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.

15 December 2008
ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

See inside cover page for obtaining further soft or hard copies.

15 December 2008
## Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 March 2008</td>
<td>pratapL</td>
<td>Added errata from Mozilla (<a href="http://www.mozilla.org/js/language/E262-3-errata.html">http://www.mozilla.org/js/language/E262-3-errata.html</a>).</td>
</tr>
<tr>
<td>26 May 2008</td>
<td>pratapL</td>
<td>Updated to introduce the following: Strict mode, infrastructure for the introduction of getters/setters, and improvements to String, Date, Array, Function, and Object. Removed the notions of equating and joining.</td>
</tr>
<tr>
<td>9 June 2008</td>
<td>pratapL</td>
<td>Added Array generics (§15.4.4.14 to 15.4.4.22). Added support for “abc”[0] (§8.6.2.1.2). Reverted the change to separate internal properties on functions. (§8.6.2) Changed Unicode Version to 4.0.</td>
</tr>
<tr>
<td>11 June 2008</td>
<td>pratapL</td>
<td>Updated 15.9.1.13 (Date Time string format) calling out that it is not required for all the fields in the ISO date format to be present. Integrated first draft of JSON changes. Applied styles uniformly for NOTEs.</td>
</tr>
<tr>
<td>14 June 2008</td>
<td>pratapL</td>
<td>Incorporated basic placeholders for Decimal.</td>
</tr>
<tr>
<td>15 June 2008</td>
<td>pratapL</td>
<td>Incorporated updated JSON exposition.</td>
</tr>
<tr>
<td>24 June 2008</td>
<td>pratapL</td>
<td>All changes related to introduction of internal [[Extensible]] property and updating the statics on Object: 4.3.3 – abbreviated the definition of object. 4.3.24 – added clarifying note: function contains executable code. 4.3.25 to 4.3.28 – added definitions for property, method, attribute, own property, inherited property. 8.6.1 – changed title text to add ‘and Descriptors’; Changed first and second para to introduce clarifying text – introduced Ddesc and Pdesc formally. 8.6.2 – introduced [[Extensible]], removed [[Dynamic]], restored allowances for host objects, and added clarifying section text. 8.6.2.1.3 – updated to use [[Extensible]]. 8.6.2.2 – updated to use [[Extensible]]; steps that said ‘throw something’ now say ‘throw TypeError’. 8.6.2.5 – step that said ‘throw something’ now says ‘throw TypeError’. 10.1.8 – property attributes for each non-negative integer arg specified. 13.2.1, 13.2.2 – updated steps to use [[Extensible]]. 15.2.3.15 – object statics specified. 15.2.4 onwards – called out the initial value of the internal [[Extensible]] property.</td>
</tr>
<tr>
<td>28 June 2008</td>
<td>pratapL</td>
<td>7.8.4, Annex A – introduced ‘LineContinuation’ to account for string literals with line terminators.</td>
</tr>
<tr>
<td>29 June 2008</td>
<td>pratapL</td>
<td>4.2 onwards – changed Writeable to Wrappable. 4.2.2 – removed “Strict and Non-Strict Modes” and replaced with “Language Subset Selection” (placeholder). 4.3.7 – introduced built-in constructor. 4.3.24 – fixed nature of a function. 4.3.29 – added Built-in Method. 7.8.5 – minor language cleanup. 8.5 – fixed reference to sections 9.5 and 9.6. 8.6.1 onwards – Property Attributes; changed [[Dynamic]] to [[Flexible]]. introduced default values for the attributes, changed ‘Strict’ flag to ‘Throw’ flag, changed [[SetOwnProperty]] to [[DefineOwnProperty]], added [[ThrowablePut]]. 8.6.2.2 – removed ‘Strict’ flag from [[Put]]. 8.6.2.2.1 – introduced [[ThrowablePut]]. 8.6.2.7 to 8.6.2.10 – added new internal properties. 8.10 – introduced whole new section. 9.10 – introduced IsCallable. 12.2 – introduced const into the grammar.</td>
</tr>
<tr>
<td>Date</td>
<td>Author</td>
<td>Changes</td>
</tr>
<tr>
<td>-------------</td>
<td>---------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>21 March 2008</td>
<td>pratapL</td>
<td>4 - updated the Scheme reference to R6R5.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.1.4 - fixed reference to Syntactic grammar.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.8.4 - clarified <code>NonEscapeCharacter</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.6.2.1 - introduced a <code>[]</code> syntax for Unicode escape sequences.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.1.2.3 - fixed reference to <code>StrDecimalLiteral</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.2.3.2 through 15.2.3.14 - changed first step to throw a Type Error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>exception if the first parameter was not of type Object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.3.2.15 - introduced <code>Object.keys static method</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.3.5.4 - introduced the “name” property.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.3.5.5 - introduced the “parameters” property.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.5.4.21 - introduced <code>String.prototype.toJSON</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.6.4.4 - introduced <code>Boolean.prototype.toJSON</code>.</td>
</tr>
<tr>
<td>30 June 2008</td>
<td>pratapL</td>
<td>11.7.5 – introduced syntactic sections for getters/setters in object literals</td>
</tr>
<tr>
<td>1 July 2008</td>
<td>pratapL</td>
<td>Integrated Unicode changes.</td>
</tr>
<tr>
<td>2 July 2008</td>
<td>pratapL</td>
<td>[Integrated changes from Allen]</td>
</tr>
</tbody>
</table>
|             |         | 7.8.5 - Backed out change requiring early reporting of invalid regex literals;
|             |         | removed additional language requiring that each such literal represents a single unique object.  |
|             |         | Added optional trailing comma in object initialisers                  |
|             |         | 11.1.5 Fixed reference to section 15.9.1.15                          |
|             |         | Introduction of “usage subset cautious”, and all restrictions (7.9.1.1, 10.1.3.1, 10.1.6.1, 10.1.8.1, 10.2.2.1, 11.3.1.1, 11.4.1.1, 11.4.5.1, 11.13.1.1, 11.13.2.1, 12.0.1, 12.6.4.1, 12.10.1, 15.1.2.1.1)  |
|             |         | 12.15.0 introduced `ConstantStatement`.                               |
|             |         | 15.5.3.3 – introduced `String.prototype.toJSON`.                      |
| 4 July 2008  | pratapL | 7.2 – added `<NEL>` as a white space character.                       |
|             |         | 7.5.2 – reverted the change that made `eval` a keyword                |
|             |         | 7.8.4 – allowed embedded LineTerminator in string literals provided they are not followed by white space.  |
|             |         | 8 – introduced Property Descriptor and Property Identifier as new types.  |
|             |         | 8.6.1 – introduced `[[Const]]` as a property attribute.               |
|             |         | 8.6.2.1, 8.6.2.3, 8.6.2.9, 8.6.2.10 – updated the algorithms to account for `[[Const]]`.  |
|             |         | 10.1.2, 10.1.3, 12, 12.1, 12.2, 12.5, 12.6, 12.6.1, 12.6.2, 12.6.3, 12.6.4, 12.10, 12.15 - introduced `Const` statements, lexical scoping of constants and function declarations, restricting declarations from use as “pseudo blocks”.  |
|             |         | 12.0.1 – deleted this section and added 12.1.1 instead.               |
|             |         | 15.13.2 – the Decimal constructor can no longer be called as a function; |
|             |         | added a clarification that it does not support the internal `[[Call]]` method. |
|             |         | 15.13.5.17, 15.13.5.20, 15.13.5.30, 15.13.5.31, 15.13.5.35, 15.13.5.41 – made `mc` an optional argument.  |
|             |         | 15.13.5.19 – made `roundingMode` an optional argument.                |
|             |         | Annex C – newly added.                                                |
|             |         | Added a ‘Draft’ watermark to main content section (section 4), and ‘saved date’ in the footer of the page.  |
|             |         | Regenerated TOC.                                                      |
| 14 July 2008 | pratapL | 15.1.2.3 through 15.2.3.14 - changed first step to throw a TypeError exception if the first parameter was not of type `Object`.  |
|             |         | 15.3.2.15 - introduced `Object.keys static method`.                   |
|             |         | 15.3.5.4 - introduced the “name” property.                            |
|             |         | 15.3.5.5 - introduced the “parameters” property.                      |
|             |         | 15.5.4.21 - introduced `String.prototype.toJSON`.                     |
|             |         | 15.6.4.4 - introduced `Boolean.prototype.toJSON`.                     |
15.7.4.8 - introduced Number.prototype.toJSON
A.1 - added NonEscapeCharacter, and productions for u { } to the lexical grammar.

15 July 2008
pratapL
8.6.2 - changed → to → in internal properties table.
10.1.2 - removed the ability to include UseSubsetDirective in the code of a function body passed as a string to the Function constructor.
10.2.4 - new section.
12.1 - various tweaks to match changes to section 10.
12.6.4 - made for-in ignore expressions whose values are either null or undefined rather than throwing a TypeError.
12.14 - update catch semantics to match changes to sections 10 and 12.1.
15.2.3.3 - moved getOwnProperty to 15.2.3.4.
15.2.3.4 - eliminated getOwnProperty method.
15.2.3.5 - added not stating that implicit string properties, corresponding to character positions are not included in the returned property descriptor.
15.2.3.5 - corrected algorithm to correctly deal with shadowed inherited properties.
15.2.3.5 - eliminated method getProperties, replaced with getOwnPropertyNames.
15.2.3.8 - changed name of second argument.
15.2.3.9 - reinstated optional second argument to Object.create.
15.2.3.10 - through 15.2.3.13 - defined a length property for the function.
A.5 - sync Program grammar in appendix with that in 14.
Throughout most in section 10: changed the term "variable object" to "environment object" and the term "variable instantiation" to "environment binding instantiations".
Major overhaul of section 10 to better accomodate block scoped const and function declarations.

4 Aug 2008
pratapL
Incorporated the following based on the Oslo review:
4 - updated the Scheme reference to the IEEE standard.
7.8.4, A.1 - reverted the addition of u { UnicodeHexEscapeSequence }, the UnicodeHexEscapeSequence production, and the elaboration on its CV.
7.2-7.3, A.1 - restored <LS> and <PS> as line termination characters. Made <CR><LF> a line terminator.
7.8.4, A.1 - restored original definition of NonEscapeCharacter.
7.8.5 - call out scan time reporting of invalid RegExp literals.
8.6.2 - changed term "procedural property" to "accessor property", removed stray mention of [[Identity]].
8.6.2.1 - changed references to Result(4) to Result(5).
8.6.2.10 - removed redundant call to [[GetOwnProperty]](O, P) in step 5.
10.1.2.3 - updated this class number to be 10.1.2.1.
15.5.4.21, 15.6.4.4, 15.7.4.8 - added 'key' as a parameter to the toJSON functions.
15.12.2 - updated step 5.b.i to call V.toJSON(KEY); updated step 5.c.i to call replacer.call(object, KEY, V).
8.6.2.4 - corrected and refactored algorithm.
8.6.2.9 - made any attribute change to a "flexible" data property acceptable.
8.10.1 - inlined IsValidDescriptor into ToPropertyDescriptor. Removed other references and deleted definition causing renumber of other definitions.
8.10.5 - (FromPropertyDescriptor) changed undefined arg result and deleted bogus step 8.
8.10.6 - (ToPropertyDescriptor) corrections in steps 1 and 9b.

15 December 2008
<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 December 2008</td>
<td></td>
<td>15.5.3.3 – reverted the addition of String.uniqueIdentifier.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.2.3.4 - renamed Object.getProperty to Object.getPropertyDescriptor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.2.3.11, 15.2.3.13 renamed Object.const to Object.freeze and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Object.isConst to Object.isFrozen.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.2.3.12, 15.2.3.15 Added Object.nonExtensible and Object.isExtensible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>functions.</td>
</tr>
<tr>
<td>15 December 2008</td>
<td></td>
<td>11.6.1, 11.6.2 - incorporated changes to the Addition and Subtraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operators to handle Decimal.</td>
</tr>
<tr>
<td>11 August 2008</td>
<td>pratapL</td>
<td>11.3.1, 11.3.2, 11.4.4, 11.4.5, 11.4.6, 11.4.7, 11.5, 11.8.1, 11.8.2, 11.8.3, 11.8.4 - incorporated changes to introduce support for Decimal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.2.3.2 - deleted Object.getOwnPropertyDescriptor; renumbered the remaining sections under 15.2.3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.2.3.3 - renamed Object.getOwnPropertyDescriptor to Object.getOwnPropertyDescriptor</td>
</tr>
<tr>
<td>18 August 2008</td>
<td>pratapL</td>
<td>11.3.1, 11.3.2, 11.4.4, 11.4.5, 11.4.6, 11.4.7, 11.5, 11.6.1, 11.6.2, 11.8.1, 11.8.2, 11.8.3, 11.8.4, 11.8.5, 11.9.3 - incorporated changes to these operators' support for Decimal to use the (yet to be defined) Form() and ToDouble() &quot;specification helper functions&quot;.</td>
</tr>
<tr>
<td>11 August 2008</td>
<td>pratapL</td>
<td>11.3.1, 11.3.2, 11.4.4, 11.4.5, 11.4.6, 11.4.7, 11.5, 11.6.1, 11.6.2, 11.8.1, 11.8.2, 11.8.3, 11.8.4, 11.8.5, 11.9.3 - incorporated changes to these operators' support for Decimal to use the (yet to be defined) Form() and ToDouble() &quot;specification helper functions&quot;.</td>
</tr>
<tr>
<td>01 September 2008</td>
<td>pratapL</td>
<td>11.3.3 – Fixed a typo in the step 4 of the algorithms for get/set property name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11.3.1, 11.3.2, 11.4.4, 11.4.5, 11.4.6, 11.4.7, 11.5, 11.6.1, 11.6.2, 11.8.1, 11.8.2, 11.8.3, 11.8.4, 11.8.5, 11.9.3, and 11.9.6 - incorporated Decimal related changes; these revert the introduction of the Form() and ToDouble() specification helper functions, and introduces calls to instanceof and Decimal functions. Handling of NaNs in comparisons has been changed to make the result more consistent with ES3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Renamed the [[Flexible]] attribute to [[Configurable]] and updated all references to it.</td>
</tr>
<tr>
<td>22 September 2008</td>
<td>pratapL</td>
<td>Decimal is now a primitive, with a wrapper.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3.30, 4.3.31, 4.3.32 - added new sections for Decimal,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3.2, 9.9, 9.10, 11.3.1, 11.3.2, 11.4.3, 11.4.4, 11.4.5, 11.4.6, 11.4.7, 11.5, 11.6.1, 11.6.2, 11.8.1, 11.8.2, 11.8.3, 11.8.4, 11.8.4, 11.8.3, 11.9.3, 11.9.6, 15.1.2.4, 15.1.2.5, 15.13.4.2 - updated all relevant operators.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.13.4.1 - deleted Decimal.valueOf()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.13.4.2, through 15.13.4.9 – introduced new statics on Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.13.5.14, 15.13.5.15 – introduced Infinity and NaN on Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.13.5.16 through 15.13.5.43 - deleted these functions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.9.1.1 - introduced restriction in cautious subset for automatic semicolon insertion that results in dead code to detect a SyntaxError.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8.10.5 - minor correction of spelling mistake (DescObj changed to Desc).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.3.4.5 - incorporated algorithmic steps for Function.prototype.bind</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.9.4.4 - introduced Date.now()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.12.1 - introduced restriction in JSON.parse that key strings within an object must be unique.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C.1.1 - updated restriction (1) according to 7.9.1.1; added restrictions (15) and (16).</td>
</tr>
</tbody>
</table>
4 - corrected minor spelling.
4.3.3, 4.3.24, 4.3.25, 4.3.28, 4.3.30 - added clarifications in the definitions of Function, Property, Own Property, and Built-in Method.

5.2 - added illustrative numbering for the algorithm steps.

7.1 - corrected minor spelling.

8.6.1 - Table 1: removed [[Const]]. Simplified description for [[Writable]].
Table 3: removed [[Const]]. Made default values for [[Writable]], [[Enumerable]] and [[Configurable]] false.

8.6.2 - restored original text that said that the description of the internal properties was for “native” ECMAScript objects. Updated second column header to be “Value Domain”, and updated second column for every internal property.

8.6.2.1 - corrected step numbering in steps 5, 7.
8.6.2.3 - updated the algorithm in steps 2, 7, and added a clarifying note.
8.6.2.8 - updated the algorithm introducing steps 2 through 8.
8.6.2.9, 8.6.2.10 - updated the algorithms removing all mention of [[Const]].

8.7 - restored original text on the non-provision within the spec for functions (built-in, and user defined) returning a reference.

8.10 - clarifications regarding absent fields, and object literal syntax for Property Descriptors.

8.10.1, 8.10.2, 8.10.3 - updated the algorithms to account for absent fields.
8.10.4 - updated algorithm removing the checks for “Unspecified”.
8.10.5 - updated algorithm removing the checks for “Unspecified”, and corrected step numbering in steps 13d, 15, 15d.

9.10 - clarified the result of IsCallable on Object.
10.1.2 - clarified exclusions for “lexical block code”.
10.1.2.1 - added to the classification of “unrestricted” code.

12.1 - corrected minor spelling.
12.2 - clarified interaction between Block and VariableDeclaration, fixed step 4 in the production VariableDeclaration : Identifier Initialiser.

12.6.4 - corrected steps 6, 8 in the first algorithm, and step 9 in the second algorithm.
12.6.4.1 - corrected the replacement steps mentioning the right Results.
12.10.1 - made the syntax error to be mandatory.
12.15 - clarified semantics around access to consts.

13.2 - replaced mention of “properties” with “parameters”.

4 - updated “use subset …” to “use strict …”
15.2.3.15 - corrected step numbering for the algorithm.
15.3.5.2 - updated [[Enumerable]] to be false.

15.3.5.5 - removed this section (it was about “parameters”).
15.4.3.2 - clarified definition of isArray.
15.10.7.5 - updated [[Writable]] to be true.

15.12.1 - corrected step 2 to mention T instead of I.

13 Oct 2008 pratapL

The “cautious” subset is now “strict” mode.
4, 4.2.2, 10.1.3.1, 10.1.6.1, 10.1.8.1, 10.2.2.1, 11.3.1.1, 11.4.1.1, 11.13.1.1, 12.1.1, 12.10.1, 15.1.2.1.1, C.1.1 - renamed “cautious” to “strict”
7.9.1.1 - removed this section (this was related to semicolon insertion in strict mode.
10.1.6.1, 10.1.8.1, 10.2.2.1, 11.3.1.1, 11.3.2.1, 11.4.1.1, 11.4.4.1, 11.4.5.1.
<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- swapped the order for steps 2, 3, and updated references to these step numbers in the rest of the algorithm; with this, the value of the MemberExpression will determine what function gets called even before the Arguments have been evaluated (and potentially caused side effects).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- deleted NOTE 1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- clarified scoping for variable statements occurring inside a FunctionExpression.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- removed the initial part of the second last paragraph that talked about the mechanics and order of enumeration of properties being implementation dependent. Added a clarification that newly added properties during enumeration are not visited in the active enumeration.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- clarified the “initial” value of Function.prototype’s [[Extensible]] property.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- removed all “this” coercion:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- both apply and call will no longer attempt to coerce “this” to an object, or to the global object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- the “this” parameter, if it is passed in, is used as-is to invoke the callback; if the this parameter is not passed in, undefined as used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- updated item 5 calling out removal the this coercion.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- fixed minor typo in item 8, updated item 11 calling out that deleting an unbound reference should throw in strict mode, fixed minor typo in item 14.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Annex D - changed [[class]] to [[Class]] in the third paragraph.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Annex E - added a clarification regarding RegExps in the second paragraph.</td>
</tr>
<tr>
<td>03 Nov 2008</td>
<td>pratapL</td>
<td>- fixed minor typo in item 8, updated item 11 calling out that deleting an unbound reference should throw in strict mode, fixed minor typo in item 14.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Annex D - changed [[class]] to [[Class]] in the third paragraph.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Annex E - added a clarification regarding RegExps in the second paragraph.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- fixed step 3 in the algorithms to test for IsCallable(Result(2)).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- made “this” coercion logic conditional on the strictness of the function.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- incorporated revised formulation for Function.prototype.bind.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- introduced [[Call]] and [[Construct]] semantics for bind.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- fixed a section reference to [[ThrowablePut]].</td>
</tr>
</tbody>
</table>
15.4.5.1 - clarified own/inherited property usage in step 14.
15.12 - introduced JSON grammar.
15.12.1 - revised pseudo-code for JSON.parse
15.12.2 - revised pseudo-code for JSON.stringify

4.2.2 - changed section title to ‘The Strict Variant of ECMAScript’
10.3.2.1, 10.4.2.1, 11.4.1.1, 11.13.1.1, 11.13.2.1, 12.3.1.1, 12.6.4.1, 12.10.1,
15.1.2.1.1 - changed section title to ‘Strict Mode Restrictions’
4.3.25 - moved text from section 10.1.1 here; renumbered subsequent subsections.
11.3.1.1, 11.3.2.1, 11.4.4.1, 11.4.5.1 - deleted these sections on usage subset restriction.
13 - function bodies can also now have a use strict directive.

Introduced the notion of Lexical Environments.
8.6.2 - renamed “scope chain” to “lexical environment” throughout the document starting from this point onwards.
8.7 - clarified the components of a “Reference” and their access (abstract operations).
8.7.1 - updated GetValue in terms of the revised abstract operations on References.
8.7.2 - PutValue no longer takes a ‘Throw’ parameter. Updated algorithm in terms of revised abstract operations on References.
10 - incorporated changes to the entire section to introduce the notion of Lexical Environments. Deleted 10.2.4 (Lexical Block Code).
11.2.3, 11.4.1, 11.4.3, 11.13.1.1, 11.13.2.1 - updated algorithm in terms of the revised abstract operations on References.
11.3.1, 11.3.2, 11.4.4, 11.4.5, 11.13.1 - updated algorithm in terms of revised PutValue.
12.2 - updated description for the VariableStatement, and the evaluation of the VariableDeclaration : Identifier Initialiser production.
12.10 - revised the algorithm in terms of the Lexical Environment and environment records.
12.14 - revised the algorithm in terms of the Lexical Environment and environment records. In ES3.1 this is the primary use case for the notion of lexical environments.
15.1.5.2 - added this subsection to call out JSON as a property on the Global Object.

Removed all mention of ConstantStatement
12, 12.15, A.4 - deleted all mention of ConstantStatement and relevant productions.

Introduced “debugger;” as a statement form.
12, 12.15, A.4 - introduced DebuggerStatement.

15.3.2.1 - removed the second last step that was adding the “parameters” property. This property is no longer supported on function instances.
C.1.1 - removed mention of when an arguments property is initialized.
D - added mention of the swapping of steps in 11.2.3.

07 Nov 2008 pratapL
11.9.7 - introduced this new section for the internal Same Value comparison function.
15.4.4.14, 15.4.4.15 - replaced ‘===' comparison of searchElement with a call to the SameValue function.
15.9.4.2 - corrected minor spelling
15.12.1, 15.12.2 - added suitable notation to indicate optional arguments in the function signatures; replaced the the test for type being “function” with IsCallable(); replaced check for the Class by actually testing the [[Class]] property.

10.3.3 - corrected minor spelling.
10.1.1, 8.10.2, 8.10.3, 8.10.4, 8.10.5, 10.2.2, 10.2.2.1, 10.2.2.2, 10.2.2.3, 10.2.2.4, 10.3.1, 10.3.2, 10.3.3, 15.5.4.14, 15.10.2.5, 15.10.2.6, 15.10.8.2, 15.10.2.15, 15.12.1, 15.12.2, 15.12.2 - Replaced inconsistent use (w.r.t. section 4.3.25) of “internal function” with “abstract operation” in relevant places

15.2.3.3, 15.2.3.4, 15.2.3.8, 15.2.3.9, 15.2.3.10 - removed all mention of “the static”
15.2.3.5, 15.2.3.6, 15.2.3.7 - renamed “method” to “function”.
15.2.4.5, 15.2.4.7 - updated algorithm to use [[GetOwnProperty]].

7.5.3 - moved ‘const’ back to the FutureReservedWords lists and added a note alluding to the future use of ‘const’, ‘let’, and ‘yield’.
8.11 - added this section as a place holder for the Environment Record Type.
9.11 - Moved SameValue algorithm here from 11.9.7
12 - added a note regarding FunctionDeclaration within a SubStatement.
Annex C, D, E - added a note saying these need to be updated.
15.2.4.1 through 15.2.4.7 - All methods that make use of their ThisValue now account for the possibility that they are being invoked (via Function.prototype.apply or Function.prototype.call) with a non-object value passed as the thisArg.
15.2.4.3 – introduced pseudo-code for toLocaleString.

Incorporated updates for Decimal
7.8.3 - introduced DecimalLiteral, and clarified rounding in the case of Decimals.
8.5 - defined the Decimal Type (alongside the Number Type).
9.2 - defined ToDecimal conversion (alongside the ToNumber conversion).
9.3.1 - defined the ToDecimal conversion as applied to Strings (alongside the ToNumber conversion).
9.6.1 - defined the ToNumber conversion as applied to the Decimal Type (alongside the Number Type).
11.3.1, 11.3.2, 11.4.4, 11.4.5, 11.4.7, 11.6.1, 11.6.2, 11.9.3, 11.9.6, 15.1.2.4, 15.1.2.5 - explicitly call out the method (as defined in IEEE 754-2008) to apply:
11.8.1, 11.8.2, 11.8.3, 11.8.4 - removed special casing for Decimal.
11.8.5 - added step 21 for Decimal
15.13.2.1 - introduced Decimal ([value] )
15.13.3.1 - clarified semantics for the constructor
15.13.4.1 through 15.13.4.9 - moved from the earlier sections 15.13.5.7 through 15.13.5.15
Deleted getExponent (was section 15.13.4.5), reduce (was section 15.13.4.8), quantize (was section 15.13.4.9), roundingMode (was section 15.13.5.3).
Introduced rescale (now section 15.13.4.15).

Regenerated TOC
<table>
<thead>
<tr>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 Dec 2008</td>
<td>pratapL</td>
<td>4.2.2 - clarified strict mode restriction for eval code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10.4.2.1 - minor editorial corrections.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reverted the Statement/SubStatement distinction.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12, 12.1, 12.1.1, 12.5, 12.6, 12.6.1, 12.6.2, 12.6.4, 12.10, A4 -</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reverted the Statement/SubStatement distinction because it isn’t needed in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the absence of lexical blocks in ES3.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.5.1, 2nd paragraph in 12.6, 12.11.1 - introduced new strict mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td>restrictions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C.1.1 - introduced strict mode restriction number 12.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Annex D - introduced 4th paragraph clarifying semantics for nested</td>
</tr>
<tr>
<td></td>
<td></td>
<td>variable declarations.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.2.1 - reverted to original form (except section reference in step 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>because the added step is already taken care of in 10.4.3.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Annex E - called out incompatibilities related to handling of &lt;NEL&gt;,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;ZWSP&gt;, &lt;BOM&gt;, &lt;CR&gt;&lt;LF&gt;, and the called out the requirement for scan-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>time reporting of RegExp constructor errors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reverted introduction of Decimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.2, 4.3.2, 7.8.3, 8.5, 9.3, 9.3.1, 9.8.1, 9.11, 11.3.1, 11.3.2, 11.4.4,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11.4.5, 11.4.6, 11.4.7, 11.5, 11.6.1, 11.6.2, 11.8.5, 11.9.3, 11.9.6,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.1.2.4, 15.1.2.5, 15.13 (including all subsections) - reverted all</td>
</tr>
<tr>
<td></td>
<td></td>
<td>special casing for Decimal.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3.32, 4.3.33, 4.3.34 - deleted these sections.</td>
</tr>
<tr>
<td>08 Dec 2008</td>
<td>pratapL</td>
<td>Regenerated TOC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.2.2 - cleaned up the 3rd paragraph that introduced strict mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.3.19 - cleaned up type definitions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.8.5 - reverted the change that said a RegExp literal evaluates to a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value of the RegExp type.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.2.25, 4.3.31 - renamed “Internal Function” to “Built-in” Function,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and updated definition of “Built-in Method” [trac 416, 417].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.3, 7.8.4 - updated LineTerminator production to match &lt;CR&gt;, and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>clarified the appearance of &lt;LS&gt; and &lt;PS&gt; in String literals [trac 412].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.8.5, Annex A.1 - updated RegExp literal grammar to accept /// as a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>valid literal [trac 419].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7.5, Annex A.1 - removed ReservedWord and Identifier from the Token</td>
</tr>
<tr>
<td></td>
<td></td>
<td>production [trac 426].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 - large reorganization of this section [trac 420, 421, 422, 425].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 - large reorganization of this section [trac 427].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.2.3.14, 15.12.2 - removed the “fast” parameter from Object.keys and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>updated its calls [trac 415].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Regenerated TOC.</td>
</tr>
<tr>
<td>15 Dec 2008</td>
<td>pratapL</td>
<td>4.3.31 - Removed all mention of “internal function” [trac 417].</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11.4.3 - clarified host objects masquerading as “function”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16 - updated the list of runtime errors that an implementation must</td>
</tr>
<tr>
<td></td>
<td></td>
<td>treat as syntax errors [trac 438].</td>
</tr>
</tbody>
</table>
7.3, Annex A.1 - fixed typo introduced when updating the `LineTerminator` production to match `<CR>`; we now have the `LineTerminatorSequence` production [trac 412].

8.6.2 - introduced `[[FormalParameters]]` as an internal property in Table 5. It is then used in 10.6, 13.2.
8.6.2 - introduced `[[Code]]` as an internal property in Table 5. It is then used in 10.4.3, 13.2.
8.6.2 - added clarifications in the description for the internal properties in Table 5.
8.12 - throughout this section explicitly added “internal” when referring to internal properties/methods.

9.10, 9.11 - mentioned that the arguments to these abstract operators must be ECMAScript language values.

10 - reorganization of this section continued 10.2.1, 10.2.1.1, 10.2.1.2 - minor clarifications.
10.2.2.1, 10.2.2.3, 10.2.3, 10.3 - cleanup.
10.4 - renamed “Entering and Execution Context” to Establishing an Execution Context” and moved it to this section number.
10.5 - moved the Arguments Object to this section number.
10.6 - moved the Declaration Binding Instantiation to this section number.

11.1.4, 11.1.5 - cleanup of Array Initialiser and Object Initialiser algorithms.
11.2.1 - clarify handling of Unicode escape sequences in `CallExpression`.
11.4.1 - cleanup of algorithm pseudo-code.

Deletion of sections calling out strict mode restrictions
11.4.1.1 - deleted this section, and folded it into a NOTE in 11.4.1.
11.13.1.1 - deleted this section, and folded it into a NOTE in 11.13.1.
12.1.1.1 - deleted this section.

13, 13.2, 13.2.1, 13.2.2 - cleanup of algorithm pseudo-code.
15.1.2.1, 15.2.3.2 through 15.2.3.14, 15.3.2.1 - cleanup of algorithm pseudo-code.
15.4.4.11 - clarification on the behaviour of the `sort`.

Regenerated TOC.