Proposal for Improving Internationalization Support in ECMAScript 2.0

Version 0.3

Friday, January 15, 1999

This document represents the current working set of recommendations from the internationalization subcommittee as to what changes should go into the core ECMAScript language to better support both internationalization and Unicode.

Pasted in below is the current working draft (1/14/99) of the standard. The changes we are proposing are highlighted by change bars. Insertions and deletions are also shown. (In a few spots, change marks from the draft we're basing this draft on bled through.) We've taken out the text of all sections of the standard we're not proposing to change. When necessary, we've left in section headings to preserve their numbering.

This proposal assumes the presence of a specification for a larger, more flexible library of internationalization functions that is outside the core language. We intend this to be a separate proposal that will either end up as an annex to this standard, or as a separate non-normative technical report. It's unclear what the appropriate way to refer to this in the standard is (especially when we don't yet have specific function names we can refer to).

When appropriate, explanatory text explaining why we're proposing a certain change, or how it got to be the way it is in this document, are inserted into the text in italic type.

1 Scope

2 Conformance

3 Normative 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), including amendments 1 through 9 and technical corrigendum 1.

ISO/IEC 646.IRV:1991 -- Information Processing -- ISO 7-bit Coded Character Set for Information Interchange.

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

Unicode Inc. (1998), Unicode Technical Report #8: The Unicode Standard™, Version 2.1. http://www.unicode.org/unicode/reports/tr8.html

Unicode Inc. (1998), Draft Unicode Technical Report #15: Unicode Normalization Forms, revision #10 (12/16/88). http://www.unicode.org/unicode/reports/tr15/tr15-10.html

ANSI/IEEE Std 754-1985: IEEE Standard for Binary Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, New York (1985).

4 Overview

5 Notational Conventions

6 Source Text

ECMAScript source text is represented as a sequence of characters ~~representable using~~in the Unicode ~~version 2.0~~ character encoding, version 2.1 or later, using the UTF-16 transformation format.

> *We're proposing to update the standard to reference Unicode 2.1 rather than Unicode 2.0 everywhere it references a particular version of Unicode. Unicode 2.1 is the last major revision of the standard; it includes many corrections to the character database, and adds two new characters, one of which is the Euro symbol.*
>
> *The purpose of adding "UTF-16" above is to back up the reference to "16-bit values" below. Other than that, the exact bit patterns used aren't all that important—they do become important, however, when you're talking about characters in a String.*

SourceCharacter **::**
any Unicode character

Except within comments and string literals, every ECMAScript program shall consist of only characters from the first 128 Unicode ~~characters~~ code points (that is, the first half of row zero). Other Unicode characters may appear only within comments and string literals. In string literals, any Unicode character may also be expressed as a Unicode escape sequence consisting of six characters from the first 128 characters, namely **\u** plus four hexadecimal digits. Within a comment, such an escape sequence is effectively ignored as part of the comment. Within a string literal, the Unicode escape sequence contributes one character to the string value of the literal.

> *The rationale behind the change of "characters" to "code points" above is to get rid of the repetition of the word "characters" in that sentence. In addition, we thought "first 128 characters" might be a little ambiguous.*

Although the characters in an ECMAScript program are Unicode characters, they are treated as independent 16-bit values with none of the context-dependent interpretation specified in the Unicode standard. Such values are often called "code points". The Unicode standard refers to code points as "coded character data elements". Throughout this International standard the terms "character" and "code point" are understood to mean "coded character data element".

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

7 Lexical Conventions

8 Types

8.1 The Undefined Type

8.2 The Null Type

8.3 The Boolean Type

8.4 The String Type

The String type is used to represent textual data in a running ECMAScript program and consists of the set of all finite ordered sequences of zero or more 16-bit Unicode characters (more properly referred to as code points; section 6). Each character is regarded as occupying a position within the sequence. These positions are identified by nonnegative integers. The leftmost character (if any) is at position 0, the next character (if any) at position 1, and so on. The length of a string is the number of distinct positions within it. The empty string has length zero and therefore contains no characters.

Text stored in a String must be treated as a sequence of 16-bit unsigned integers, regardless of how the characters are actually storeed. This format is the standard Unicode format, and is equivalent to the UTF-16 transformation format in the ISO/IEC 10646 universal character encoding.

> *The reason for specifying the transformation format is that UTF-8, UTF-16, and UCS-4 would all have different character positions within the same string—surrogate pairs in UTF-16 are single characters in UCS-4, and single characters in UCS-4 can occupy one, two, three, or four bytes in UTF-8. The idea here is to standardize things here so that offsets within Strings are always interpreted as if the String were stored in UTF-16.*

Because Unicode can represent single characters with a sequence of code points (a "combining character sequence"), and UTF-16 represents characters with code-point values above 0xFFFF as a sequence of two 16-bit code points (a "surrogate pair"), it is possible for string operations to affect the semantics of the characters in the string (by breaking up combining character sequences and surrogate pairs). All of the built-in String operations in this standard (except toUpperCase() and toLowerCase() treat a String as a sequence of 16-bit integer values and make no attempt to protect the integrity of these sequences.

> *This was part of the long discussion we had on the subject of normalization at the 1/11 meeting. We all agreed that all core-language functions that operate on Strings (except toUpperCase() and toLowerCase()) completely ignore the semantics of the characters they operate on. Strings are treated as sequences of undifferentiated 16-bit values. There is nothing to stop a user from splitting up a combining character sequence or concatenating a string that begins with a combining diacritic to the end of another string (which changes the interpretation of that string's last character). No automatic normalization happens, and the programmer is responsible for not messing up his strings.*

It is also possible to represent the same string with several different sequences of code points. The built-in String operations do not take this into account when comparing Strings for equality. Because of this, we recommend that text coming into the environment of a running ECMAScript program from the host environment be converted to Unicode Normalized Form C (canonical composition). Usually this process would occur at the same time as incoming text is converted from its original character encoding into Unicode. However, string literals in ECMAScript source code should never be normalized, and none of the built-in String functions automatically normalize either their inputs or their results.

> *This was also part of the long discussion on normalization at the 1/11 meeting. We all agreed to relax the requirement that strings be normalized to a recommendation (in the case of strings coming in through the embedding), and to explicitly **not** allow automatic normalization of strings within the running program's environment.*

~~1.5~~8.5 The Number Type

8.6 The Object Type

8.7 The Reference Type

8.8 The List Type

8.9 The Completion Type

9 Type Conversion

9.1 ToPrimitive

9.2 ToBoolean

9.3 ToNumber

9.3.1 ToNumber Applied to the String Type

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

StringNumericLiteral **:::**
StrWhiteSpace$_{opt}$
StrWhiteSpace$_{opt}$ *StrNumericLiteral* *StrWhiteSpace*$_{opt}$

StrWhiteSpace **:::**
StrWhiteSpaceChar *StrWhiteSpace*$_{opt}$

StrWhiteSpaceChar **:::**
<TAB>
<SP>
<FF>
<VT>
<CR>
<LF>
All Unicode white-space characters (i.e., everything defined in the Unicode character database as belonging to categories Zs, Zp, or Zl).

> *We discussed extended the definition of DecimalDigit similarly, and we also considered adding proper interpretation of other Unicode mathematical operators (particularly the minus sign), but decided to leave all that to the internationalization package. The expanded definition of white space here is to make sure that all white space is treated as white space rather than merely as illegal characters.*

StrNumericLiteral **:::**
StrDecimalLiteral
+ *StrDecimalLiteral*
- *StrDecimalLiteral*
HexIntegerLiteral

StrDecimalLiteral **:::**
`Infinity`
DecimalDigits **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$
. *DecimalDigits* *ExponentPart*$_{opt}$
DecimalDigits *ExponentPart*$_{opt}$

DecimalDigits **:::**
DecimalDigit
DecimalDigits DecimalDigit

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

ExponentPart **:::**
ExponentIndicator SignedInteger

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

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

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

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

Some differences should be noted between the syntax of a *StringNumericLiteral* and a *NumericLiteral* (section **Error! Reference source not found.**):

- A *StringNumericLiteral* may be preceded and/or followed by whitespace and/or line terminators.
- A StringNumericLiteral may not use octal notation.
- 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 whitespace is converted to **+0**.

The conversion of a string to a number value is similar overall to the determination of the number value for a numeric literal (section **Error! Reference source not found.**), 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, ideally using IEEE 754 round-to-nearest mode, to a representable value of the number type.

- The MV of *StringNumericLiteral* **:::** (an empty character sequence) is 0.
- The MV of *StringNumericLiteral* **:::** *StrWhiteSpace* is 0.
- The MV of *StringNumericLiteral* **:::** $StrWhiteSpace_{opt}$ *StrNumericLiteral* $StrWhiteSpace_{opt}$ is the MV of *StrNumericLiteral*, no matter whether whitespace is present or not.
- The MV of *StrNumericLiteral* **:::** *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* **:::** **+** *StrDecimalLiteral* is the MV of *StrDecimalLiteral*.
- The MV of *StrNumericLiteral* **:::** **-** *StrDecimalLiteral* is the negative of the MV of *StrDecimalLiteral*. (Note that if the MV of *StrDecimalLiteral* is 0, the negative of this MV is also 0. The rounding rule described below handles the conversion of this signless mathematical zero to a floating-point **+0** or **−0** as appropriate.)
- The MV of *StrNumericLiteral* **:::** *HexIntegerLiteral* is the MV of *HexIntegerLiteral*.
- The MV of *StrDecimalLiteral* **:::** **Infinity** is 10^{10000} (a value so large that it will round to **+∞**).
- The MV of *StrDecimalLiteral* **:::** *DecimalDigits***.** is the MV of *DecimalDigits*.
- The MV of *StrDecimalLiteral* **:::** *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 *DecimalDigit*s.
- The MV of *StrDecimalLiteral* **:::** *DecimalDigits***.** *ExponentPart* is the MV of *DecimalDigits* times 10^{e}, where *e* is the MV of *ExponentPart*.

- The MV of *StrDecimalLiteral* **:::** *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 *DecimalDigit*s and *e* is the MV of *ExponentPart*.
- The MV of *StrDecimalLiteral* **:::** **.** *DecimalDigits* is the MV of *DecimalDigits* times 10^{-n}, where *n* is the number of characters in *DecimalDigit*s.
- The MV of *StrDecimalLiteral* **:::** **.** *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times 10^{e-n}, where *n* is the number of characters in *DecimalDigit*s and *e* is the MV of *ExponentPart*.
- The MV of *StrDecimalLiteral* **:::** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *StrDecimalLiteral* **:::** *DecimalDigits ExponentPart* is the MV of *DecimalDigits* times 10^{e}, where *e* is the MV of *ExponentPart*.
- The MV of *DecimalDigits* **:::** *DecimalDigit* is the MV of *DecimalDigit*.
- The MV of *DecimalDigits* **:::** *DecimalDigits DecimalDigit* is (the MV of *DecimalDigits* times 10) plus the MV of *DecimalDigit*.
- The MV of *ExponentPart* **:::** *ExponentIndicator SignedInteger* is the MV of *SignedInteger*.
- The MV of *SignedInteger* **:::** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* **:::** **+** *DecimalDigits* is the MV of *DecimalDigits*.
- The MV of *SignedInteger* **:::** **-** *DecimalDigits* is the negative of the MV of *DecimalDigits*.
- The MV of *DecimalDigit* **:::** **0** or of *HexDigit* **:::** **0** is 0.
- The MV of *DecimalDigit* **:::** **1** or of *HexDigit* **:::** **1** is 1.
- The MV of *DecimalDigit* **:::** **2** or of *HexDigit* **:::** **2** is 2.
- The MV of *DecimalDigit* **:::** **3** or of *HexDigit* **:::** **3** is 3.
- The MV of *DecimalDigit* **:::** **4** or of *HexDigit* **:::** **4** is 4.
- The MV of *DecimalDigit* **:::** **5** or of *HexDigit* **:::** **5** is 5.
- The MV of *DecimalDigit* **:::** **6** or of *HexDigit* **:::** **6** is 6.
- The MV of *DecimalDigit* **:::** **7** or of *HexDigit* **:::** **7** is 7.
- The MV of *DecimalDigit* **:::** **8** or of *HexDigit* **:::** **8** is 8.
- The MV of *DecimalDigit* **:::** **9** or of *HexDigit* **:::** **9** is 9.
- The MV of *HexDigit* **:::** **a** or of *HexDigit* **:::** **A** is 10.
- The MV of *HexDigit* **:::** **b** or of *HexDigit* **:::** **B** is 11.
- The MV of *HexDigit* **:::** **c** or of *HexDigit* **:::** **C** is 12.
- The MV of *HexDigit* **:::** **d** or of *HexDigit* **:::** **D** is 13.
- The MV of *HexDigit* **:::** **e** or of *HexDigit* **:::** **E** is 14.
- The MV of *HexDigit* **:::** **f** or of *HexDigit* **:::** **F** is 15.
- The MV of *HexIntegerLiteral* **:::** **0x** *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* **:::** **0X** *HexDigit* is the MV of *HexDigit*.
- The MV of *HexIntegerLiteral* **:::** *HexIntegerLiteral HexDigit* is (the MV of *HexIntegerLiteral* times 16) plus the MV of *HexDigit*.

Once the exact MV for a string numeric literal has been determined, it is then rounded to a value of the Number type. If the MV is 0, then the rounded value is +0 unless the first non-whitespace character in the string numeric literal is '-', in which case the rounded value is –0. Otherwise, the rounded value must be the number value for the MV (in the sense defined in section 8.5), unless the literal includes a StrDecimalLiteral 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 (either 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)).

~~1.4~~9.4 ToInteger

9.5 ToInt32: (Signed 32 Bit Integer)

9.6 ToUint32: (Unsigned 32 Bit Integer)

9.7 ToUint16: (Unsigned 16 Bit Integer)

9.8 ToString

9.9 ToObject

10 Execution Contexts

11 Expressions

11.1 Primary Expressions

11.2 Left-Hand-Side Expressions

11.3 Postfix Expressions

11.4 Unary Operators

11.5 Multiplicative Operators

11.6 Additive Operators

11.7 Bitwise Shift Operators

11.8 Relational Operators

11.8.1 The Less-than Operator (<)

11.8.2 The Greater-than Operator (>)

11.8.3 The Less-than-or-equal Operator (<=)

11.8.4 The Greater-than-or-equal Operator (>=)

11.8.5 The Abstract Relational Comparison Algorithm

The comparison *x* < *y*, where *x* and *y* are values, produces **true**, **false**, or **undefined** (which indicates that at least one operand is **NaN**). Such a comparison is performed as follows:

1. Call ToPrimitive(*x*, hint Number).
2. Call ToPrimitive(*y*, hint Number).
3. If Type(Result(1)) is String and Type(Result(2)) is String, go to step 16. (Note that this step differs from step 7 in the algorithm for the addition operator **+** in using *and* instead of *or*.)
4. Call ToNumber(Result(1)).
5. Call ToNumber(Result(2)).
6. If Result(4) is **NaN**, return **undefined**.
7. If Result(5) is **NaN**, return **undefined**.
8. If Result(4) and Result(5) are the same number value, return **false**.
9. If Result(4) is **+0** and Result(5) is –**0**, return **false**.
10. If Result(4) is –**0** and Result(5) is **+0**, return **false**.
11. If Result(4) is **+∞**, return **false**.
12. If Result(5) is **+∞**, return **true**.

13. If Result(5) is –∞, return **false**.
14. If Result(4) is –∞, return **true**.
15. If the mathematical value of Result(4) is less than the mathematical value of Result(5)—note that these mathematical values are both finite and not both zero—return **true**. Otherwise, return **false**.
16. If Result(2) is a prefix of Result(1), 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 Result(1) is a prefix of Result(2), return **true**.
18. Let *k* be the smallest nonnegative integer such that the character at position *k* within Result(1) is different from the character at position *k* within Result(2). (There must be such a *k*, for neither string is a prefix of the other.)
19. Let *m* be the integer that is the Unicode encoding for the character at position *k* within Result(1).
20. Let *n* be the integer that is the Unicode encoding for the character at position *k* within Result(2).
21. If *m* < *n*, return **true**. Otherwise, return **false**.

NOTE The comparison of strings uses a simple lexicographic ordering on sequences of Unicode 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 ~~2.0~~ specification. This means that strings which are canonically equal according to the Unicode standard could test as unequal. In effect, this algorithm assumes both strings are already in normalized form.

11.8.6 The instanceof operator

11.8.7 The in operator

11.9 Equality Operators

11.9.1 The Equals Operator (==)

11.9.2 The Does-not-equals Operator (!=)

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* == ToNumber(*y*).
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: **`"" + a == "" + b`**.

Numeric comparison can be forced by: **`a - 0 == b - 0`**.

Boolean comparison can be forced by: **`!a == !b`**.

The equality operators maintain the following invariants:

1. **A != B** is equivalent to **!(A == B)**.

2. **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.

The comparison of strings uses a simple lexicographic ordering on sequences of Unicode 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. This means that strings which are canonically equal according to the Unicode standard could test as unequal. In effect, this algorithm assumes both strings are already in normalized form.

~~Comparison of strings uses a simple equality test on sequences of Unicode 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 2.0 specification.~~

This function merely calls toString(): the locale parameter is ignored.

This is here so that all ECMAScript objects have a toLocaleString() method. The default behavior is to call toString(); this is only different for Number, Date, and Array objects (and possibly user-defined objects). Having toLocaleString() on everything allows Array to produce localized results without having to worry about what specifically is stored in it.

The locale parameter is here to allow the user to specify a locale if the external I18N package is present (in which case that library defines the type of this parameter); otherwise, this parameter is ignored by the toLocaleString() functions in the core language.

~~15.2.4.3~~15.2.4.4 **Object.prototype.valueOf ( )**

15.2.4.5 Object.prototype.hasProperty(V)

15.2.4.6 Object.prototype.hasDelegate(V)

15.2.4.7 Object.prototype.propertyIsEnumerable(V)

15.2.5 Properties of Object Instances

15.3 Function Objects

15.4 Array Objects

15.4.1 The Array Constructor Called as a Function

15.4.2 The Array Constructor

15.4.3 Properties of the Array Constructor

15.4.4 Properties of the Array Prototype Object

15.4.4.1 Array.prototype.constructor

15.4.4.2 Array.prototype.toString ( )

The elements of this object are converted to strings using their toString() methods, and these strings are then concatenated, separated by comma characters. The result is the same as if the built-in **`join`** method were invoked for this object with no argument.

15.4.4.3 Array.prototype.toLocaleString(locale)

The elements of this object are converted to strings using their toLocaleString(locale) methods. The locale parameter to this function is passed through to all of the other toLocaleString() calls. These strings are then concatenated, separated by comma characters.

~~15.4.4.3~~15.4.4.4 ~~Array.prototype.concat(array1, array2, ...)~~Array.prototype.concat ([item1 [, item2 [, ...]]])

~~15.4.4.4~~15.4.4.5 Array.prototype.join (separator)

~~15.4.4.5~~15.4.4.6 Array.prototype.pop ( )

~~15.4.4.6~~15.4.4.7 Array.prototype.push (item1, item2, …)

~~15.4.4.7~~15.4.4.8 Array.prototype.reverse ( )

~~15.4.4.8~~15.4.4.9 Array.prototype.shift ( )

~~15.4.4.9~~15.4.4.10 Array.prototype.slice (start [,end])

~~15.4.4.10~~15.4.4.11 Array.prototype.sort (comparefn)

~~15.4.4.11~~15.4.4.12 ~~Array.prototype.splice(start, deleteCount, item1, item2, ...)~~Array.prototype.splice (start, deleteCount [, item1 [, item2 [, …]]])

~~15.4.4.12~~15.4.4.13 ~~Array.prototype.unshift(item1, item2, ...)~~Array.prototype.unshift ([item1 [, item2 [, …]]])

15.4.5 Properties of Array Instances

15.5 String Objects

15.5.1 The String Constructor Called as a Function

15.5.2 The String Constructor

15.5.3 Properties of the String Constructor

The value of the internal [[Prototype]] property of the String constructor is the Function prototype object.

15.5.4 Properties of the String Prototype Object

15.5.4.1 String.prototype.constructor

15.5.4.2 String.prototype.toString ( )

15.5.4.3 String.prototype.valueOf ( )

15.5.4.4 String.prototype.charAt (pos)

15.5.4.5 String.prototype.charCodeAt (pos)

15.5.4.6 String.prototype.concat (string1, string2, …)

15.5.4.7 String.prototype.indexOf (searchString, position)

15.5.4.8 String.prototype.lastIndexOf (searchString, position)

15.5.4.9 String.prototype.slice (start [, end])

15.5.4.10 String.prototype.split (separator)

15.5.4.11 String.prototype.substring (start [, end])

15.5.4.12 String.prototype.toLowerCase (locale)

~~Returns a string equal in length to the length of the result of converting this object to a string.~~If this object is not 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 in the String are converted one by one. The result of each conversion is the original character, ~~Every character of the result is equal to the corresponding character of the string,~~ unless that character has a Unicode 2.1~~0~~ lowercase equivalent, in which case the lowercase equivalent is used instead. ~~(The canonical Unicode 2.0 case mapping shall be used, which does not depend on implementation or locale.)~~ The result is intended to be produced in a locale-sensitive way. ECMAScript implementations supporting the external internationalization package should vector through to an appropriate routine in that package, passing the locale parameter through. Implementations not supporting the internationalization package should ignore the locale parameter and produce a result that is correct for the system default locale, if possible, and fall back on the canonical Unicode case mappings otherwise.

> *We're specifically proposing changing toLowerCase() and toUpperCase() to have language-specific behavior (the canonical Unicode case mappings get you about 95% of the way there, so this isn't that hard). Again, if the I18N package is there, these functions are supposed to vector through to it and it defines the interpretation of that parameter. If the I18N package isn't there, the parameter is ignored, and you either get locale-specific behavior if it's available from the host environment or fall back on the canonical mappings.*

Note that 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.

15.5.4.13 String.prototype.toUpperCase (locale)

~~Returns a string equal in length to the length of the result of converting this object to a string.~~ If this object is not a String, it is converted to a String. The characters in that String are converted one by one to upper case. The result is a string value, not a string object.

~~Every character of the result is equal to the corresponding character of the string,~~ The characters in the String are converted one by one. The result of each conversion is the original character, unless that character has a Unicode 2.1~~0~~ uppercase equivalent, in which case the uppercase equivalent is used instead. ~~(The canonical Unicode 2.0 case mapping shall be used, which does not depend on implementation or locale.)~~ The result is intended to be produces in a locale-sensitive way. ECMAScript implementations supporting the external internationalization package should vector through to an appropriate routine in that package, passing the locale parameter through. Implementations not supporting the internationalization package should ignore the locale parameter and produce a result that is correct for the system default locale, if possible, and fall back to the canonical Unicode case mappings otherwise.

Note that 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.

15.5.4.14 String.prototype.normalize()

Converts the String to Unicode Normalized Form C (canonical decomposition followed by canonical composition) as described in Unicode Technical Report #15. The result is semantically equal to the input string. If the input string was already properly normalized, the result is identical to the input string.

The purpose of this function is to convert a String into its canonical representation. Normalizing two strings before comparing them ensures that two strings that the Unicode standard says are equal are actually treated as equal by the program. If an ECMAScript implementation produces normalized text as part of the process of translating from the native character encoding to Unicode, programmers should not have to worry about normalization unless they're dealing with malformed Unicode text (e.g., sequences that begin with combining characters, or sequences containing isolated surrogate characters) or deliberately create non-normalized strings themselves.

> *We had originally taken normalize() out of this proposal and put it in the external library, but there seemed to be widespread consensus that it was needed in the core language. The data tables necessary to do normalization are currently about 40K in size (although it might be possible to do better), so we may need some kind of escape hatch here.*

Produces a String that represents the value of the Number formatted according to the conventions of the specified locale. If the implementation supports the external internationalization package, this function should vector through to an appropriate routine in that package, passing the locale parameter through. Otherwise, it should ignore the locale parameter and produce a result that is formatted correctly for the system default locale. If this is not possible, this function is permitted to fall back on toString().

15.9.5.25 Date.prototype.setUTCMilliseconds (ms)

15.9.5.26 Date.prototype.setSeconds (sec [, ms])

15.9.5.27 Date.prototype.setUTCSeconds (sec [, ms])

15.9.5.28 Date.prototype.setMinutes (min [, sec [, ms]])

15.9.5.29 Date.prototype.setUTCMinutes (min [, sec [, ms]])

15.9.5.30 Date.prototype.setHours (hour [, min [, sec [, ms]]])

15.9.5.31 Date.prototype.setUTCHours (hour [, min [, sec [, ms]]])

15.9.5.32 Date.prototype.setDate (date)

15.9.5.33 Date.prototype.setUTCDate (date)

15.9.5.34 Date.prototype.setMonth (month [, date])

15.9.5.35 Date.prototype.setUTCMonth (month [, date])

15.9.5.36 Date.prototype.setFullYear (year [, month [, date]])

15.9.5.37 Date.prototype.setUTCFullYear (year [, month [, date]])

15.9.5.38 Date.prototype.toLocaleString (locale)

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 appropriate to the geographic or cultural locale. If an ECMAScript implementation supports the external internationalization package, this function should vector through the appropriate function in that library, passing the locale parameter along. Otherwise, it should ignore the locale parameter and produce a result appropriate to the current system default locale. Failing this, it is permissible to fall back on toString().

15.9.5.39 Date.prototype.toUTCString ( )

15.9.5.40 Date.prototype.toGMTString ( )

15.9.6 Properties of Date Instances

16 Errors

17 Compatibility

Our original proposal also suggested some extensions (and some explanatory language) to the section on regular expressions. Since this working draft of the standard doesn't have the sections on regular expressions, we had nowhere to introduce these changes. Presumably, the whole regular-expression syntax is the subject of a different proposal that hasn't been integrated into the working draft yet—the proper place for our changes is there.