Proposal for Improving Internationalization Support in ECMAScript 2.0

Version 1.0
Thursday, April 29, 1999

This document contains the final set of recommendations for adding better i18n support in ECMAScript. All of the changes are (as far as I know) content-agreed in this form. I've removed all the commentary from this draft.

Recommended Changes

Pasted in below are the relevant parts of the 1/14/99 working draft 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 on bled through.) We've taken out the text of all sections of the standard we're not proposing to change.

2 Conformance

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

A conforming implementation of this International standard shall interpret characters in conformance with the Unicode Standard, Version ~~2.0~~2.1 or later, and ISO/IEC 10646-1 with either UCS-2 or UTF-16 as the adopted encoding form, implementation level 3. If the adopted ISO/IEC 10646-1 subset is not otherwise specified, it is presumed to be the BMP subset, collection 300. If the adopted encoding form is not otherwise specified, it presumed to be the UTF-16 encoding form.

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

A conforming implementation of ECMAScript is permitted to support program syntax not described in this specification. In particular, a conforming implementation of ECMAScript is permitted to support program syntax that makes use of the "future reserved words" listed in section **Error! Reference source not found.** of this specification.

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~~plus its amendments and corrigenda.

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.

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

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

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. The text is expected to have been normalized to Unicode Normalized Form C (canonical composition), as described in Unicode Technical Report #15. Conforming ECMAScript implementations are not required to perform any normalization of text, or behave as though they were performing normalization of text, themselves.

SourceCharacter **::**
any Unicode character

~~Except within comments and string literals, every ECMAScript program shall consist of only characters from the first 128 Unicode characters (that is, the first half of row zero). Other Unicode characters may appear only within comments and string literals.~~ ECMAScript source text can contain any of the Unicode characters: all Unicode white-space characters are treated as whitespace, and all Unicode line/paragraph separators are treated as line separators. Non-Latin Unicode characters are allowed in identifiers, string literals, and comments.

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. This effectively limits valid source text to the characters in the Basic Multilingual Plane of ISO 10646: letters, digits, and whitespace defined outside the BMP are not guaranteed to be treated as letters, digits, and whitespace by the source-code parser.

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

~~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

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.

Syntax

InputElement **::**
WhiteSpace
LineTerminator
Comment
Token

7.1 Unicode Format-Control Characters

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

However, the format control characters shall have no lexical significance in the context of an ECMAScript program. These characters are transparent to both the lexical analyzer and the parser and do not affect either's interpretation of the source text. The only exception is that format characters within a string literal do contribute a character to the string.

~~7.1~~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:

~~*Unicode*~~ *Code Point Value*	*Name*	*Formal Name*
\u0009	Tab	<TAB>
\u000B	Vertical Tab	<VT>
\u000C	Form Feed	<FF>
\u0020	Space	<SP>
\u00a0	No-break space	<NBSP>
category = "Zs"	All Unicode "space separators"	<USP>

Syntax

WhiteSpace **::**

<TAB>
<VT>
<FF>
<SP>
<NBSP>
<USP> [i.e., any Unicode character in category Zs (“space separator”), according to the Unicode standard

~~7.2~~7.3 Line Terminators

Like whitespace characters, line terminator characters are used to improve source text readability and to separate tokens (indivisible lexical units) from each other. However, unlike whitespace characters, line terminators have some influence over the behaviour of the syntactic grammar. In general, line terminators may occur between any two tokens, but there are a few places where they are forbidden by the syntactic grammar. A line terminator cannot occur within any token, not even a string. Line terminators also affect the process of automatic semicolon insertion (section **Error! Reference source not found.**).

The following characters are considered to be line terminators:

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

Syntax

LineTerminator **::**

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

~~7.3~~7.4 Comments

~~7.4~~7.5 Tokens

~~7.5~~7.6 Identifiers

Description

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

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

Escape sequences are also permitted in identifiers, where they contribute a single code-point value to the identifier. An escape sequence can not be used to put a character into an identifier that would otherwise be illegal.

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 normalized form C before it reaches the compiler.

~~An identifier is a character sequence of unlimited length, where each character in the sequence must be a letter, a decimal digit, an underscore (_) character, or a dollar sign ($) character, and the first character may not be a decimal digit. ECMAScript identifiers are case sensitive: identifiers whose characters differ in any way, even if only in case, are considered to be distinct. The dollar sign ($) character is intended for use only in mechanically generated code.~~

Syntax

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

IdentifierName **::**
~~*IdentifierLetter*~~*IdentifierStart*

~~*IdentifierName IdentifierLetter*~~
IdentifierName ~~*DecimalDigit*~~*IdentifierPart*

IdentifierStart **::**
UnicodeLetter [any Unicode character in the categories "Uppercase letter (Lu)", "Lowercase letter (Ll)", "Titlecase letter (Lt)", "Modifier letter (Lm)", "Other letter (Lo)", or "Letter number (Nl)".]
$
_
UnicodeEscapeSequence [so long as it represents one of the other characters specified here]

IdentifierPart **::**
IdentifierStart
UnicodeCombiningMark [any Unicode character in the categories "Non-spacing mark (Mn)" or "Combining spacing mark (Mc)"]
UnicodeDigit [any Unicode character in the category "Decimal number (Nd)"
UnicodeConnectorPunctuation [any character in the category "Connector punctuation (Pc)"
UnicodeEscapeSequence [so long as it represents one of the other characters specified here]

UnicodeEscapeSequence **::**
\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

~~*IdentifierLetter* :: 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**~~
~~**$ _**~~

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

8 Types

8.4 The String Type

The String type is the set of all finite ordered sequences of zero or more 16-bit ~~Unicode characters (more properly referred to as code points; section 6)~~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 Unicode code point (see section 6). Each ~~character~~ element is regarded as occupying a position within the sequence. These positions are ~~identified by~~indexed with nonnegative integers. The ~~leftmost character~~first element (if any) is at position 0, the next ~~character~~ element (if any) at position 1, and so on. The length of a string is the number of ~~distinct positions~~elements (i.e., 16-bit values) within it. The empty string has length zero and therefore contains no ~~characters~~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 for toUpperCase(), toLowerCase(), and localeCompare()) treat them as sequences of undifferentiated 16-bit unsigned integers; they do not ensure the resulting string is in normalized 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 Normalized 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 we also recommend that ECMAScript source code be in Normalized Form C, string literals are guaranteed to be normalized (if source text is guaranteed to be normalized), as long as it doesn't contain any Unicode escape sequences.

9 Type Conversion

9.3.1 ToNumber Applied to the String Type

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

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

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

StrWhiteSpaceChar **:::**
<TAB>
<SP>
<NBSP>
<FF>
<VT>
<CR>
<LF>
<LS>
<PS>
any Unicode character in the category Zs, according to the Unicode character database

11 Expressions

11.8.5 The Abstract Relational Comparison Algorithm

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 that both strings are already in normalized form.

11.9.3 The Abstract Equality Comparison Algorithm

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 means that 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 normalized form.

15 Native ECMAScript Objects

15.2 Object Objects

15.2.4.3 Object.prototype.toLocaleString()

This function merely calls toString().

NOTE The first parameter to this function is likely to be used in the next version of this standard; we recommend that implementations do not use this parameter position for anything else.

15.3 Function Objects

15.4 Array Objects

15.4.4.2 Array.prototype.toString ( )

~~The elements of this object are converted to strings, and these strings are then concatenated, separated by comma characters.~~ The result of calling this function 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()

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 **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 [[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(). The first parameter passed to this function is passed unchanged to the toLocaleString() method being called.
14. Let *R* be a string value produced by concatenating *S* and Result(13).
15. Increase *k* by 1.
16. Go to step 10.

NOTE The first parameter to this function is likely to be used in the next version of this standard; we recommend that implementations do not use this parameter position for anything else.

15.5 String Objects

~~1.1.1.12~~15.5.4.12 String.prototype.toLowerCase (~~-~~)

~~Returns a string equal in length to the length of the result of converting this object to a string.~~ 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, ~~Every character of the result is equal to the corresponding character of the string,~~ unless that character has a Unicode 2.~~0~~1 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 should be derived according to the *non-locale-specific* case mappings in the Unicode character database (this explicitly includes not only the UnicodeData.txt file included as part of the standard, but also the SpecialCasing.txt file that accompanies it in Unicode 2.1.8 and later.

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.

~~1.1.1.13~~15.5.4.13 String.prototype.toUpperCase (~~-~~)

~~Returns a string equal in length to the length of the result of converting this object to a string. The result is a string value, not a string object.~~

~~Every character of the result is equal to the corresponding character of the string, unless that character has a Unicode 2.0 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.)~~

This function behaves in exactly the same way as String.prototype.toLowerCase(locale), except that characters are mapped to their *uppercase* equivalents as specified in the Unicode Character Database (both UnicodeData.txt and SpecialCasing.txt).

Because both toUpperCase() and toLowerCase() have context-sensitive behavior, the functions are not symmetrical. In other words, s.toUpperCase().toLowerCase() is not equal to s.toLowerCase().

However, s.toLowerCase().toUpperCase() is currently guaranteed by the Unicode case mappings to be equal to s.toUpperCase(). Therefore, for simple case-insensitive operations, normalize the strings using toUpperCase() instead of toLowerCase().

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.

~~1.7~~15.7 Number Objects

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 The first parameter to this function is likely to be used in the next version of this standard; we recommend that implementations do not use this parameter position for anything else.

15.9 Date Objects

15.9.5.38 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 a convenient, human-readable form appropriate to the geographic or cultural locale.~~Produces a string value that represents the Date in a human-readable form that corresponds to the conventions of the host environment's current locale. This function is implementation-dependent.

NOTE The first parameter to this function is likely to be used in the next version of this standard; we recommend that implementations do not use this parameter position for anything else.