Draft Status: Rough

2008-04-24, pratapL, creation

2008-05-01, pratapL, added String.prototype.quote; updated with comments from Doug, Adam, Brendan

Rationale of the ES3.1 String Proposal

Almost every Ajax library implements a **`trim`** function for strings. Implementations are available from YUI (html), Atlas (html), Dojo (html), Prototype (html), Base2 (html), etc. The **`trim`** function uses a regular expression to match white space, and returns a string with both the leading and trailing white space removed. It seems fairly trivial to implement, but to get a performant implementation one really has to know regexps fairly well and do some performance analysis. For e.g. see this blog entry (html). Experience with Dojo suggests that String trimming is often a hotspot in profiling.

The **`quote`** method on String is another example of a useful utility function. FireFox implements it; and several use-cases can benefit from not having to roll-their-own (for e.g. JSON).

ES3.1 proposes to codify such string "generics".

There are several methods on ES3 objects that are referred to as "generic" methods. "Generic" here means that the method's **`this`** parameter can be any object that can be coerced to a String. Therefore, it can be transferred to other kinds of objects. ES3 provides such methods on the respective objects' prototypes. If we want to add more such generic methods, where should they be added? Should they be added on the respective object's prototype? Should they be added as static methods on the respective objects, instead? Although adding it to the prototype may seem desirable it means that to invoke such a generic prototype method on an object of a different class, one must use `Function.prototype.apply` or `call`, or explicitly perform the coercsion. This can be incovenient as shown in the following example where the generic method String.prototype.charAt is called on an Array Object. Note that such coercion incurs the overhead of a copy.

```
<script>
var a = new Array('0', '1');
document.write(String.prototype.charAt.call(a, 0));
document.write(String(a).charAt(0));
</script>
```

Proposed ES4 Proposal

The proposed ES4 proposal (html) adds a static method of the same name for each prototype generic method, taking an explicit leading obj parameter that binds to this in the callee. Such "statics" are useful in scenarios where the overhead of a copy is not OK; for e.g. when used on StringLike wrappers, as the one LiveConnect makes for java.lang.String. Or other such "wrappers" that crop up when bridging between runtimes with similar (but not identical, and not quite compatible) notions of String.

ES3.1 Proposal

The goals of the ES3.1 proposal are to ensure the following:

- Compatible with ES3
- Compatible with ES4

Instead of adding the methods as statics one might add the methods on **`String.prototype`**. Would adding such generics as methods on the prototype collide with what is already provided by the libraries? Perhaps it is appropriate to allow such collisions in this case; we really don't need to consider different libraries implementing these as "collisions", but rather as agreed upon meaning; in any case, libraries typically test for the existence of such methods

before augmenting adding it themselves. Also, specifying such generics as statics does not conform to how the rest of the ES3 methods are specified.

ES3.1 proposes the following:

The **trim** function returns a string value (not a String object) with both the leading and trailing white space removed. The definition of white space (see "Definition of White Space" below) is the set of characters matched by the regular expression **/\s/** (15.10.2.12). The **trim** function is intentionally generic; "generic" here means that the method's **this** parameter can be any object that can be coerced to a string.

The **quote** function returns a quoted string value. The value is surrounded by single quotes and any interior single or double quotes are escaped (a preceding backslash is added). The **quote** function is intentionally generic; "generic" here means that the method's **this** parameter can be any object that can be coerced to a String.

These methods will be added to **String.prototype**.

These will need to be supported in proposed-ES4 too to maintain the subset relationship.

Definition of White Space

The definition of white space as used by the lexer is defined in ES3 in section 7.2.

The **trim** function available from various libaries uses the regular expression **/\s/** to match white spaces. In ES3, the class of characters matched by this regular expression is defined (15.10.2.12) as the characters that are on the right-hand side of the WhiteSpace (7.2) or LineTerminator (7.3) productions.

However, what actually gets matched as white space by the the regular expression **/\s/** differs between browsers (some even treat characers in other categories as white space), as shown by the test below:

```
<script>
// White Space : section 7.2
var whitespace = [
{s : 'foo\u0009bar', t : 'HORIZONTAL TAB'},
{s : 'foo\u000Bbar', t : 'VERTICAL TAB'},
{s : 'foo\u000Cbar', t : 'FORMFEED'},
{s : 'foo\u0020bar', t : 'SPACE'},
{s : 'foo\u00A0bar', t : 'NO-BREAK SPACE'},
{s : 'foo\u1680bar', t : 'OGHAM SPACE MARK'},
{s : 'foo\u180Ebar', t : 'MONGOLIAN VOWEL SEPARATOR'},
{s : 'foo\u2000bar', t : 'EN QUAD'},
{s : 'foo\u2001bar', t : 'EM QUAD'},
{s : 'foo\u2002bar', t : 'EN SPACE'},
{s : 'foo\u2003bar', t : 'EM SPACE'},
{s : 'foo\u2004bar', t : 'THREE-PER-EM SPACE'},
{s : 'foo\u2005bar', t : 'FOUR-PER-EM SPACE'},
{s : 'foo\u2006bar', t : 'SIX-PER-EM SPACE'},
{s : 'foo\u2007bar', t : 'FIGURE SPACE'},
{s : 'foo\u2008bar', t : 'PUNCTUATION SPACE'},
{s : 'foo\u2009bar', t : 'THIN SPACE'},
```

```
{s : 'foo\u200Abar', t : 'HAIR SPACE'},
{s : 'foo\u202Fbar', t : 'NARROW NO-BREAK SPACE'},
{s : 'foo\u205Fbar', t : 'MEDIUM MATHEMATICAL SPACE'},
{s : 'foo\u3000bar', t : 'IDEOGRAPHIC SPACE'}
];

// Line terminators : section 7.3
var lineterminators = [
{s : 'foo\u000Abar', t : 'LINE FEED OR NEW LINE'},
{s : 'foo\u000Dbar', t : 'CARRIAGE RETURN'},
{s : 'foo\u2028bar', t : 'LINE SEPARATOR'},
{s : 'foo\u2029bar', t : 'PARAGRAPH SEPARATOR'}
];

// Others : historical?
var others = [
{s : 'foo\u200Bbar', t : 'ZERO WIDTH SPACE (category Cf)'}
];

var regex = /\s/;

document.write("White space: section 7.2" + "<br>");
for (var i = 0; i < whitespace.length; i++) {
  var found = regex.test(whitespace[i].s);
  document.write(whitespace[i].t + '\t:' + found);
  document.write("<br>");
}

document.write("<br>");

document.write("Line Terminators: section 7.3" + "<br>");
for (var i = 0; i < lineterminators.length; i++) {
  var found = regex.test(lineterminators[i].s);
  document.write(lineterminators[i].t + '\t:' + found);
  document.write("<br>");
}

document.write("<br>");
```

```
document.write("Others" + "<br>");
for (var i = 0; i < others.length; i++) {
  var found = regex.test(others[i].s);
  document.write(others[i].t + '\t:' + found);
  document.write("<br>");
}
</script>
```

The table below identifies each browser's behaviour (blank cells indicate that the corresponding character is not matched as white space by **`/\s/`**).

Character	IE	Safari	FF	Opera
\u0009 : HORIZONTAL TAB	Y	Y	Y	Y
\u000B : VERTICAL TAB	Y	Y	Y	Y
\u000C : FORMFEED	Y	Y	Y	Y
\u0020 : SPACE	Y	Y	Y	Y
\u00A0 : NO-BREAK SPACE			Y	Y
\u1680 : OGHAM SPACE MARK				Y
\u180E : MONGOLIAN VOWEL SEPARATOR				
\u2000 : EN QUAD			Y	Y
\u2001 : EM QUAD			Y	Y
\u2002 : EN SPACE			Y	Y
\u2003 : EM SPACE			Y	Y
\u2004 : THREE-PER-EM SPACE			Y	Y
\u2005 : FOUR-PER-EM SPACE			Y	Y
\u2006 : SIX-PER-EM SPACE			Y	Y
\u2007 : FIGURE SPACE			Y	Y
\u2008 : PUNCTUATION SPACE			Y	Y
\u2009 : THIN SPACE			Y	Y
\u200A : HAIR SPACE			Y	Y
\u202F : NARROW NO-BREAK SPACE				Y
\u205F : MEDIUM MATHEMATICAL SPACE				
\u3000 : IDEOGRAPHIC SPACE			Y	Y
\u000A : LINE FEED OR NEW LINE	Y	Y	Y	Y
\u000D : CARRIAGE RETURN	Y	Y	Y	Y
\u2028: LINE SEPARATOR			Y	Y
\u2029 : PARAGRAPH SEPARATOR			Y	Y

\u200B : ZERO WIDTH SPACE (category "Cf")			Y	Y

In terms of commonality across browsers, only the characters `[\t, \v, \f, \n, \r, \u0020]` are matched as white space, and none of the implementations exactly follows the ES3 specification.

What should be the definition of white space that **`trim`** should use then? Should the definition of characters matched by **`/\s/`** as called out in 15.10.2.12 be updated to reflect reality? After all, library authors do not seem to have special-cased their trim implementations on a per-browser basis. On the other hand, a stricter approach would require the exact application of 15.10.2.12. In this particular case the spec-language has been clear; hence we shall prefer the latter.

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 [[Value]] 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 { DontEnum, DontDelete, ReadOnly }.

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.

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 +∞; 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`**.

Comment [pL1]: This correction is from the ES3 errata.

Deleted: 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* (*NCapturingParens* 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")`** returns **`"$1-$11,$1-$22"`**. A **`$`** in *newstring* that does not match any of the forms below is left as is.

Characters	*Replacement text*
`$$`	**`$`**
`$&`	The matched substring.
`$``	The portion of *string* that precedes the matched substring.
`$'`	The portion of *string* that follows the matched substring.
`$`*n*	The *n*th capture, where *n* is a single digit 1-9 and **`$`***n* is not followed by a decimal digit. If *n*≤*m* and the *n*th capture is **undefined**, use the empty string instead. If *n*>*m*, the result is implementation-defined.
`$`*nn*	The *nn*th capture, where *nn* is a two-digit decimal number 01-99. If *nn*≤*m* and the *nn*th capture is **undefined**, use the empty string instead. If *nn*>*m*, the result is implementation-defined.

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()`**.
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* = *separator*; otherwise let *R* = ToString(*separator*).
7. If *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 *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 *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 [[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 *cap*, go to step 10.
22. Let *i* = *i*+1.
23. Call the [[Put]] method of *A* with arguments *A*`.length` and *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 *s* (exclusive).
29. Call the [[Put]] method of *A* with arguments *A*`.length` and *T*.
30. Return *A*.
31. Call *SplitMatch*(*R*, *S*, 0) and let *z* be its MatchResult result.
32. If *z* is not **failure**, return *A*.
33. Call the [[Put]] method of *A* with arguments **`"0"`** and *S*.
34. Return *A*.

The internal helper function *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 *cap* be an empty array of captures (see 15.10.2.1).
7. Return the State (*q*+*r*, *cap*). (see 15.10.2.1)
8. Call the [[Match]] method of *R* giving it the arguments *S* and *q*, and return the MatchResult result.

The **length** property of the **split** method is **2**.

NOTE 1
*The **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.*

NOTE 2
*The **split** method ignores the value of separator **.global** for separators that are RegExp objects.*

15.5.4.15 String.prototype.substring (start, end)

The **substring** 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* 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.

15.5.4.19 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 whitespace removed. The definition of white space is the set of characters matched by the regular expression **`/\s/`** (15.10.2.12). 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*

15.5.4.20 String.prototype.quote ( )

If this object is not already a string, it is converted to a string. The result is a copy of the string except that it is surrounded by single quotes and any interior single or double quotes are escaped (a preceding backslash is added). The result is a string value, not a String object.

NOTE
*The **`quote`** 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 [[Value]] property and a **`length`** property.

The [[Value]] 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 unchanging. It has the attributes { DontEnum, DontDelete, ReadOnly }.