String

The `String` object is used to represent and manipulate a sequence of characters.

Description

Strings are useful for holding data that can be represented in text form. Some of the most-used operations on strings are to check their `length`, to build and concatenate them using the + and += string operators, checking for the existence or location of substrings with the `indexOf()` method, or extracting substrings with the `substring()` method.

Creating strings

Strings can be created as primitives, from string literals, or as objects, using the `String()` constructor:

```
const string1 = "A string primitive";
const string2 = 'Also a string primitive';
const string3 = `Yet another string primitive`;
```

```
const string4 = new String("A String object");
```

String primitives and string objects can be used interchangeably in most situations. See "String primitives and String objects" below.

String literals can be specified using single or double quotes, which are treated identically, or using the backtick character `` ` ``. This last form specifies a template literal: with this form you can interpolate expressions.

Character access

There are two ways to access an individual character in a string. The first is the `charAt()` method:

```
return 'cat'.charAt(1) // returns "a"
```

The other way (introduced in ECMAScript 5) is to treat the string as an array-like object, where individual characters correspond to a numerical index:

```
return 'cat'[1] // returns "a"
```

When using bracket notation for character access, attempting to delete or assign a value to these properties will not succeed. The properties involved are neither writable nor configurable. (See `Object.defineProperty()` for more information.)

Comparing strings

In C, the `strcmp()` function is used for comparing strings. In JavaScript, you just use the less-than and greater-than operators:

```
let a = 'a'
let b = 'b'
if (a < b) { // true
  console.log(a + ' is less than ' + b)
} else if (a > b) {
  console.log(a + ' is greater than ' + b)
} else {
  console.log(a + ' and ' + b + ' are equal.')
}
```

A similar result can be achieved using the `localeCompare()` method inherited by `String` instances.

Note that `a == b` compares the strings in `a` and `b` for being equal in the usual case-sensitive way. If you wish to compare without regard to upper or lower case characters, use a function similar to this:

```
function isEqual(str1, str2)
{
    return str1.toUpperCase() === str2.toUpperCase()
} // isEqual
```

Upper case is used instead of lower case in this function, due to problems with certain UTF-8 character conversions.

String primitives and String objects

Note that JavaScript distinguishes between `String` objects and primitive string values. (The same is true of `Boolean` and `Numbers`.)

String literals (denoted by double or single quotes) and strings returned from `String` calls in a non-constructor context (that is, called without using the `new` keyword) are primitive strings. JavaScript automatically converts primitives to `String` objects, so that it's possible to use `String` object methods for primitive strings. In contexts where a method is to be invoked on a primitive string or a property lookup occurs, JavaScript will automatically wrap the string primitive and call the method or perform the property lookup.

```
let s_prim = 'foo'
let s_obj = new String(s_prim)

console.log(typeof s_prim) // Logs "string"
console.log(typeof s_obj)  // Logs "object"
```

String primitives and `String` objects also give different results when using `eval()`. Primitives passed to `eval` are treated as source code; `String` objects are treated as all other objects are, by returning the object. For example:

```
let s1 = '2 + 2'              // creates a string primitive
let s2 = new String('2 + 2')  // creates a String object
console.log(eval(s1))         // returns the number 4
console.log(eval(s2))         // returns the string "2 + 2"
```

For these reasons, the code may break when it encounters `String` objects when it expects a primitive string instead, although generally, authors need not worry about the distinction.

A `String` object can always be converted to its primitive counterpart with the `valueOf()` method.

```
console.log(eval(s2.valueOf()))  // returns the number 4
```

Escape sequences

Escape sequences

Special characters can be encoded using escape sequences:

Escape sequence	Unicode code point
`\0`	null character (U+0000 NULL)
`\'`	single quote (U+0027 APOSTROPHE)
`\"`	double quote (U+0022 QUOTATION MARK)
`\\`	backslash (U+005C REVERSE SOLIDUS)
`\n`	newline (U+000A LINE FEED; LF)
`\r`	carriage return (U+000D CARRIAGE RETURN; CR)
`\v`	vertical tab (U+000B LINE TABULATION)
`\t`	tab (U+0009 CHARACTER TABULATION)
`\b`	backspace (U+0008 BACKSPACE)
`\f`	form feed (U+000C FORM FEED)
`\uXXXX` …where `XXXX` is exactly 4 hex digits in the range `0000` – `FFFF` ; e.g., `\u000A` is the same as `\n` (LINE FEED); `\u0021` is " ! "	Unicode code point between `U+0000` and `U+FFFF` (the Unicode Basic Multilingual Plane)
`\u{X}` … `\u{XXXXXX}` …where `X` … `XXXXXX` is 1–6 hex digits in the range `0` – `10FFFF` ; e.g., `\u{A}` is the same as `\n` (LINE FEED); `\u{21}` is " ! "	Unicode code point between `U+0000` and `U+10FFFF` (the entirety of Unicode)
`\xXX` …where `XX` is exactly 2 hex digits in the range `00` – `FF` ; e.g., `\x0A` is the same as `\n` (LINE FEED); `\x21` is " ! "	Unicode code point between `U+0000` and `U+00FF` (the Basic Latin and Latin-1 Supplement blocks; equivalent to ISO-8859-

Escape sequence	Unicode code point[1]
\x21 is !	

Long literal strings

Sometimes, your code will include strings which are very long. Rather than having lines that go on endlessly, or wrap at the whim of your editor, you may wish to specifically break the string into multiple lines in the source code without affecting the actual string contents. There are two ways you can do this.

Method 1

You can use the + operator to append multiple strings together, like this:

```
let longString = "This is a very long string which needs " +
                 "to wrap across multiple lines because " +
                 "otherwise my code is unreadable."
```

Method 2

You can use the backslash character (\ ) at the end of each line to indicate that the string will continue on the next line. Make sure there is no space or any other character after the backslash (except for a line break), or as an indent; otherwise it will not work.

That form looks like this:

```
let longString = "This is a very long string which needs \
to wrap across multiple lines because \
otherwise my code is unreadable."
```

Both of the above methods result in identical strings.

Constructor

`String()`

Creates a new `String` object. It performs type conversion when called as a function, rather than as a constructor, which is usually more useful.

Static methods

`String.fromCharCode(num1 [, ...[, numN]])`

Returns a string created by using the specified sequence of Unicode values.

`String.fromCodePoint(num1 [, ...[, numN)`

Returns a string created by using the specified sequence of code points.

`String.raw()`

Returns a string created from a raw template string.

Instance properties

`String.prototype.length`

Reflects the `length` of the string. Read-only.

Instance methods

`String.prototype.at(index)` 

Returns the character (exactly one UTF-16 code unit) at the specified `index`. Accepts negative integers, which count back from the last string character.

`String.prototype.charAt(index)`

Returns the character (exactly one UTF-16 code unit) at the specified `index`.

`String.prototype.charCodeAt(index)`

Returns a number that is the UTF-16 code unit value at the given `index`.

`String.prototype.codePointAt(pos)`

Returns a nonnegative integer Number that is the code point value of the UTF-16 encoded code point starting at the specified `pos`.

`String.prototype.concat(str [, ...strN ])`

Combines the text of two (or more) strings and returns a new string.

`String.prototype.includes(searchString [, position])`

Determines whether the calling string contains `searchString`.

`String.prototype.endsWith(searchString [, length])`

Determines whether a string ends with the characters of the string `searchString`.

`String.prototype.indexOf(searchValue [, fromIndex])`

Returns the index within the calling **`String`** object of the first occurrence of `searchValue`, or `-1` if not found.

`String.prototype.lastIndexOf(searchValue [, fromIndex])`

Returns the index within the calling **`String`** object of the last occurrence of `searchValue`, or `-1` if not found.

`String.prototype.localeCompare(compareString [, locales [, options]])`

Returns a number indicating whether the reference string `compareString` comes before, after, or is equivalent to the given string in sort order.

`String.prototype.match(regexp)`

Used to match regular expression `regexp` against a string.

`String.prototype.matchAll(regexp)`

Returns an iterator of all `regexp`'s matches.

`String.prototype.normalize([form])`

Returns the Unicode Normalization Form of the calling string value.

`String.prototype.padEnd(targetLength [, padString])`

Pads the current string from the end with a given string and returns a new string of the length `targetLength`.

`String.prototype.padStart(targetLength [, padString])`

Pads the current string from the start with a given string and returns a new string of the length `targetLength`.

`String.prototype.repeat(count)`

Returns a string consisting of the elements of the object repeated `count` times.

`String.prototype.replace(searchFor, replaceWith)`

Used to replace occurrences of `searchFor` using `replaceWith`. `searchFor` may be a string or Regular Expression, and `replaceWith` may be a string or function.

`String.prototype.replaceAll(searchFor, replaceWith)`

Used to replace all occurrences of `searchFor` using `replaceWith`. `searchFor` may be a string or Regular Expression, and `replaceWith` may be a string or function.

`String.prototype.search(regexp)`

Search for a match between a regular expression `regexp` and the calling string.

`String.prototype.slice(beginIndex[, endIndex])`

Extracts a section of a string and returns a new string.

`String.prototype.split([sep [, limit] ])`

Returns an array of strings populated by splitting the calling string at occurrences of the substring `sep`.

`String.prototype.startsWith(searchString [, length])`

Determines whether the calling string begins with the characters of string `searchString`.

`String.prototype.substring(indexStart [, indexEnd])`

Returns a new string containing characters of the calling string from (or between) the specified index (or indeces).

`String.prototype.toLocaleLowerCase( [locale, ...locales])`

The characters within a string are converted to lowercase while respecting the current locale.

For most languages, this will return the same as `toLowerCase()`.

`String.prototype.toLocaleUpperCase( [locale, ...locales])`

The characters within a string are converted to uppercase while respecting the current locale.

For most languages, this will return the same as `toUpperCase()`.

`String.prototype.toLowerCase()`

Returns the calling string value converted to lowercase.

`String.prototype.toString()`

Returns a string representing the specified object. Overrides the `Object.prototype.toString()` method.

`String.prototype.toUpperCase()`

Returns the calling string value converted to uppercase.

`String.prototype.trim()`

Trims whitespace from the beginning and end of the string. Part of the ECMAScript 5 standard.

`String.prototype.trimStart()`

Trims whitespace from the beginning of the string.

`String.prototype.trimEnd()`

Trims whitespace from the end of the string.

`String.prototype.valueOf()`

Returns the primitive value of the specified object. Overrides the `Object.prototype.valueOf()` method.

`String.prototype.@@iterator()`

Returns a new iterator object that iterates over the code points of a String value, returning each code point as a String value.

HTML wrapper methods

> **Warning:** Deprecated. Avoid these methods.
>
> They are of limited use, as they provide only a subset of the available HTML tags and attributes.

`String.prototype.anchor()`

`<a name="name">` (hypertext target)

`String.prototype.big()`

`<big>`

`String.prototype.blink()`

`<blink>`

`String.prototype.bold()`

`<b>`

`String.prototype.fixed()`

`<tt>`

`String.prototype.fontcolor()`

`<font color="color">`

`String.prototype.fontsize()`

`<font size="size">`

`String.prototype.italics()`

`<i>`

`String.prototype.link()`

`<a href="url">` (link to URL)

`String.prototype.small()`

`<small>`

`String.prototype.strike()`

`<strike>`

`String.prototype.sub()`

`<sub>`

`String.prototype.sup()`

`<sup>`

Examples

String conversion

String conversion

It's possible to use `String` as a more reliable `toString()` alternative, as it works when used on `null` and `undefined`. For example:

```
var nullVar = null;
nullVar.toString();       // TypeError: nullVar is null
String(nullVar);          // "null"

var undefinedVar;
undefinedVar.toString();  // TypeError: undefinedVar is undefined
String(undefinedVar);     // "undefined"
```

Specifications

Specification
ECMAScript Language Specification (ECMAScript) # sec-string-objects

Browser compatibility

Report problems with this compatibility data on GitHub

String	
Chrome	1
Edge	12
Firefox	1
Internet Explorer	3
Opera	3
Safari	1
WebView Android	1
Chrome Android	18