

# JS Fundamentals Lesson 2

Mohamed Emary

August 22, 2023

## Handling text — strings in JavaScript

**Template literal** is a string in which we use backtick characters ```, instead of using single or double quote marks `'` or `"`. It's like f-strings in python. It allows you to embed expressions in a string, by wrapping the expression in `${}`.

ex:

```
1  const one = "Hello, ";
2  const two = "how are you?";
3  const joined = `${one}${two}`;
4  console.log(joined); // "Hello, how are you?"
```

---

Every number has a method called `toString()` that converts it to the equivalent string.

The `Number()` function converts anything passed to it into a number.

---

Template literals respect the line breaks (You can also just use `\n` if you want) in the source code, so you can write strings that span multiple lines like this:

```
1  const output = `I like the song.
2  I gave it a score of 90%.`;
3  console.log(output);
4
5  /*
6  The output will be:
7  I like the song.
8  I gave it a score of 90%.
9  */
```

---

## JavaScript String Methods

JS is just like python in case of indexing the last index is not included.

**All string methods return a new string. They don't modify the original string.**

**Strings are immutable: Strings cannot be changed, only replaced.**

Table 1: `substr()`, `substring()`, and `slice()` Comparison

Method	Parameters	Negative Index
<code>substr()</code>	<code>start</code> , <code>length</code> <b>not end</b>	Allowed
<code>substring()</code>	<code>start</code> , <code>end</code>	Treated as zero
<code>slice()</code>	<code>start</code> , <code>end</code>	Allowed

`substr()` method is deprecated, use `substring()` or `slice()` instead.

In the 3 functions if you omit the `end` parameter it will slice out the rest of the string.

---

`replace()` function notes:

- The `replace()` method replaces only the first match
- If you want to replace all matches, **use a regular expression** with the `/g` flag set, or use the `replaceAll()` method.
- `replace()` method is case sensitive, and to replace case insensitive, use a regular expression with an `/i` flag (*insensitive*):

Important note related to regular expressions: **regular expressions are written without quotes.**

The regex begins and ends with `/` and the `g` flag means global, i.e. replace all occurrences, `i` flag means case insensitive.

Example: `console.log(my_str.replace(/this/gi, "that"));`

If you use regex with `replaceAll()` method you have to use the `g` flag, otherwise a `TypeError` is thrown.

---

Padding methods `padStart()` and `padEnd()` take **length** and **character** as parameters:  
`string.padStart(length, character)`

Strings can be indexed using `[]` or `charAt()` If no character is found, `[]` returns `undefined`, while `charAt()` returns an empty string

**strings in JS are immutable**, so you can't change a character in a string by doing something like `my_str[0] = "A";` you will not get an error, but the string will not change.

```
1 text.split(",")      // Split on commas
2 text.split(" ")      // Split on spaces
3 text.split("|")      // Split on pipe
4 text.split("")       // Split on individual characters
```

Table 2: Some Common JavaScript String Methods

Method	Parameters	Return Value
<code>length</code>	N/A	The length of the string

Method	Parameters	Return Value
<code>replace()</code>	<code>searchValue</code> , <code>replaceValue</code>	A new string with the first occurrence of <code>searchValue</code> replaced with <code>replaceValue</code> (unless <code>RegExp</code> is used)
<code>replaceAll()</code>	<code>searchValue</code> , <code>replaceValue</code>	A new string with all occurrences of <code>searchValue</code> replaced with <code>replaceValue</code> (new in ECMAScript 2021)
<code>toUpperCase()</code>	N/A	A new string with all characters converted to uppercase
<code>toLowerCase()</code>	N/A	A new string with all characters converted to lowercase
<code>concat()</code>	<code>string2</code> , <code>string3</code> , ...	A new string that concatenates the original string with one or more additional strings
<code>trim()</code>	N/A	A new string with all whitespace removed from the beginning and end of the original string
<code>trimStart()</code>	N/A	A new string with all whitespace removed from the beginning of the original string (new in ECMAScript 2021)
<code>trimEnd()</code>	N/A	A new string with all whitespace removed from the end of the original string (new in ECMAScript 2021)
<code>padStart()</code>	<code>targetLength</code> , <code>padString</code>	A new string with the original string padded with <code>padString</code> at the beginning to reach the specified <code>targetLength</code>
<code>padEnd()</code>	<code>targetLength</code> , <code>padString</code>	A new string with the original string padded with <code>padString</code> at the end to reach the specified <code>targetLength</code>
<code>charAt()</code>	<code>index</code>	The character at the specified <code>index</code>
<code>charCodeAt()</code>	<code>index</code>	The Unicode value of the character at the specified <code>index</code>
<code>split()</code>	<code>separator</code> , <code>limit</code>	An array of substrings created by splitting the original string at each occurrence of <code>separator</code> (or at most <code>limit</code> occurrences)

## JavaScript String Reference

JavaScript, methods and properties are also available to strings, because **JavaScript treats strings as objects**

look at String HTML Wrapper Methods in the page

Table 3: `indexOf()` and `repeat()` methods

Name	Description
<code>indexOf()</code>	Returns the index (position) of <b>the first occurrence</b> of a value in a string
<code>repeat()</code>	Returns a new string with a number of copies of a string

## String

**String primitives** and **string objects** share many behaviors, but have other important differences

The `eval(string_primitive)` function:

```

1 const s1 = "2 + 2";           // creates a string primitive
2 const s2 = new String("2 + 2"); // creates a String object
3 console.log(eval(s1));        // returns the number 4
4 console.log(eval(s2));        // returns the string "2 + 2"

```

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

```

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

```

### String coercion

Many built-in operations that expect strings first coerce their arguments to strings (which is largely why String objects behave similarly to string primitives). The operation can be summarized as follows:

- Strings are returned as-is.
- `undefined` turns into `"undefined"`.
- `null` turns into `"null"`.
- `true` turns into `"true"`; `false` turns into `"false"`.
- Numbers are converted with the same algorithm as `toString(10)`.
- `BigInts` are converted with the same algorithm as `toString(10)`.
- Symbols throw a `TypeError`.
- Objects are first converted to a primitive by calling its `[[toPrimitive]]()` (with `"string"` as hint), `toString()`, and `valueOf()` methods, in that order. The resulting primitive is then converted to a string. For example, if we have an object `obj` it is first converted to a primitive by calling `obj[[toPrimitive]]("string")`, then `obj.toString()`, and then `obj.valueOf()`.

**There are several ways to achieve nearly the same effect in JavaScript.**

- Template literal: ``${x}`` does exactly the string coercion steps explained above for the embedded expression.
- The `String()` function: `String(x)` uses the same algorithm to convert `x`, except that Symbols don't throw a `TypeError`, but return `"Symbol(description)"`, where `description` is the description of the Symbol.
- Using the `+` operator: `" " + x` coerces its operand to a primitive instead of a string, and, for some objects, has entirely different behaviors from normal string coercion. See its reference page for more details.

## Comparisons

Table 4: null and undefined comparison

Value	Type	Numeric Conversion	Equality Check	Strict Equality Check	Comparison
null	null	0	null, undefined only return true, other values are false	only return true, other values are false	<, <=, >, >= 0
undefined	undefined	NaN		undefined only return true, other values are false	Always false

## Conditionals

A common pattern you'll come across again and again. Any value **that is not** false, undefined, null, 0, NaN, or an empty string "" actually returns true when tested as a conditional statement, therefore you can use a variable name on its own to test whether it is true, or even that it exists (that is, it is not undefined.)

So for example:

```
1 let cheese = "Cheddar";
2
3 if (cheese) {
4   console.log("Yay! Cheese available for making cheese on toast.");
5 } else {
6   console.log("No cheese on toast for you today.");
7 }
```

As you see from the above code, cheese is used as a conditional statement and it returns true because it is neither false, undefined, null, 0, NaN, or an empty string "".

See the two examples at the end of the article.