Usage of the backtick character (`) in JavaScript?

198

In JavaScript, a backtick† seems to work the same as a single quote. For instance, I can use a backtick to define a string like this:

```
var s = `abc`;
```

Is there any way in which the behavior of the backtick actually differs from that of a single quote?

40

† Note that among programmers, "backtick" is one name for what is more generally called the grave accent. Programmers also sometimes use the alternate names "backquote" and "backgrave". Also, on Stack Overflow and elsewhere, other common spellings for "backtick" are "back-tick" and "back tick".

javascript backticks single-quotes template-strings backquote

edited Jan 8 at 13:39
DavidRR
10.1k 11 65 140

asked Dec 28 '14 at 15:59
caesarwang
1,278 2 10 11

6 Answers

227

This is a feature called **template literals**.

They were called "template strings" in prior editions of the ECMAScript 2015 specification.

Template literals are supported by Firefox 34, Chrome 41, and Edge 12 and above, but not by Internet Explorer.

- Examples: http://te20wiki.calculist.org/es6/template-strings/

```
var a = 123, str = `---
   a is: ${a}
---`;
console.log(str);
```

Output:

```
---
   a is: 123
---
```

What is more important, they can contain not just a variable name, but any Javascript expression:

```
var a = 3, b = 3.1415;

console.log(`PI is nearly ${Math.max(a, b)}`);
```

edited Oct 5 '18 at 17:34

community wiki
12 revs, 9 users 57%
try-catch-finally

chromestatus.com/feature/4743002513735680 – epascarello Oct 26 '15 at 23:40

2 Are there any viable polyfils for this given the lack of support for it? – Alexander Dixon Jul 7 '16 at 18:16

3 @AlexanderDixon, no you can't polyfill this language feature in the classical sense, though you might use templates from Underscore or lodash for variables in strings in combination with multilining strings using arrays: `["a", "b"].join(""); // both string elements written in new lines` . But apart from this one might use a "transpiler" like Babel to convert ES6+ to ES5 – try-catch-finally Jul 7 '16 at 19:29

2 *Tagged template literals* using backticks! This is valid and works well: `alert`1`` . – Константин Ван Jul 29 '16 at 20:59

2 @kiki it looks like the script language is a variant of ECMAScript. Google App scripts do not support ECMAScript 2015 features obviously. I was unable to find an official specification what language they're using. – try-catch-finally Mar 30 at 11:53

+50

```
let person = {name: 'RajiniKanth', age: 68, greeting: 'Thalaivaaaa!' };

let usualHtmlStr = "<p>My name is " + person.name + ",</p>\n" +
  "<p>I am " + person.age + " old</p>\n" +
  "<strong>\"" + person.greeting + "\" is what I usually say</strong>";

let newHtmlStr =
 `<p>My name is ${person.name},</p>
  <p>I am ${person.age} old</p>
  <p>"${person.greeting}" is what I usually say</strong>`;

console.log(usualHtmlStr);
console.log(newHtmlStr);
```

As you can see, we used the `` ` `` around a series of characters, which are interpreted as a string literal, but any expressions of the form `${..}` are parsed and evaluated inline immediately.

One really nice benefit of interpolated string literals is they are allowed to split across multiple lines:

```
var Actor = {"name": "RajiniKanth"};

var text =
`Now is the time for all good men like ${Actor.name}
to come to the aid of their
country!`;
console.log(text);
// Now is the time for all good men
// to come to the aid of their
// country!
```

Interpolated Expressions

Any valid expression is allowed to appear inside `${..}` in an interpolated string literal, including function calls, inline function expression calls, and even other interpolated string literals!

```
function upper(s) {
  return s.toUpperCase();
}
```

```
// A very WARM welcome
// to all of you READERS!
```

Here, the inner `` `${who}s` `` interpolated string literal was a little bit nicer convenience for us when combining the who variable with the `"s"` string, as opposed to `who + "s"`. Also to keep an note is an interpolated string literal is just lexically scoped where it appears, not dynamically scoped in any way

```
function foo(str) {
  var name = "foo";
  console.log(str);
}
function bar() {
  var name = "bar";
  foo(`Hello from ${name}!`);
}
var name = "global";
bar(); // "Hello from bar!"
```

Using the template literal for the HTML is definitely more readable by reducing the annoyance.

The plain old way:

```
'<div class="' + className + '">' +
  '<p>' + content + '</p>' +
  '<a href="' + link + '">Let\'s go</a>'
'</div>';
```

With ECMAScript 6:

```
`<div class="${className}">
  <p>${content}</p>
  <a href="${link}">Let's go</a>
</div>`
```

- Your string can span multiple lines.

We can also tag a template string, when a template string is tagged, the literals and substitutions are passed to function which returns the resulting value.

```
function myTaggedLiteral(strings) {
  console.log(strings);
}

myTaggedLiteral`test`; //["test"]

function myTaggedLiteral(strings, value, value2) {
  console.log(strings, value, value2);
}
let someText = 'Neat';
myTaggedLiteral`test ${someText} ${2 + 3}`;
//["test", ""]
// "Neat"
// 5
```

We can use the spread operator here to pass multiple values. The first argument—we called it strings—is an array of all the plain strings (the stuff between any interpolated expressions).

We then gather up all subsequent arguments into an array called values using the `... gather/rest operator` , though you could of course have left them as individual named parameters following the strings parameter like we did above `(value1, value2, etc.)` .

```
function myTaggedLiteral(strings, ...values) {
  console.log(strings);
  console.log(values);
}

let someText = 'Neat';
myTaggedLiteral`test ${someText} ${2 + 3}`;
//["test", ""]
// "Neat"
// 5
```

The argument(s) gathered into our values array are the results of the already evaluated interpolation expressions found in the string literal. A tagged string literal is like a processing step after the interpolations are evaluated, but before the final string value is compiled, allowing you more control over generating the string from the literal. Let's look at an example of creating a re-usable templates

```
const ActorTemplate = templater`<article>
  <h3>${'name'} is a Actor</h3>
  <p>You can find his movies at ${'store'}.</p>

</article>`;

function templater(strings, ...keys) {
  return function(data) {
    let temp = strings.slice();
    keys.forEach((key, i) => {
      temp[i] = temp[i] + data[key];
    });
    return temp.join('');
  }
};

const myTemplate = ActorTemplate(Actor);
console.log(myTemplate);
```

Raw Strings

Our tag functions receive a first argument we called strings, which is an array. But there's an additional bit of data included: the raw unprocessed versions of all the strings. You can access those raw string values using the `.raw` property, like this:

```
function showraw(strings, ...values) {
  console.log(strings);
  console.log(strings.raw);
}
showraw`Hello\nWorld`;
```

As you can see, the raw version of the string preserves the escaped `\n` sequence, while the processed version of the string treats it like an unescaped real new-line. ECMAScript 6 comes with a built-in function that can be used as a string literal tag: `String.raw(..)`. It simply passes through the raw versions of the strings:

```
console.log(`Hello\nWorld`);
/* "Hello
World" */
```


Peter Mortensen
14.2k 19 88 115


Thalaivar
16.2k 4 51 65

1 Great answer! Minor comment, in your Tagged Template Literals section, I believe the two example array outputs for `myTaggedLiteral`test ${someText} ${2 + 3}`;` should be `//["test ", " "]` (i.e. not trimmed strings). – Michael Krebs Feb 11 '17 at 12:11

2 Scrolled down to see the account of the author, was not disappointed! Good explanation. xD – varun Nov 8 '18 at 13:06

Good explanation and wide coverage, thank you. Just wanted to add that there is also a good overview on the Mozilla developer site Template literals (Template strings) which covers some extra aspects. – Dev Ops Apr 10 at 7:35

19

Backticks (`) are used to define template literals. Template literals are a new feature in ES6 to make working with strings easier.

Features:

- we can interpolate any kind of expression in the template literals.
- They can be multi-line.

Note: we can easily use single quotes (') and double quotes (") inside the backticks (`).

Example:

```
var nameStr = `I'm "Rohit" Jindal`;
```

To interpolate the variables or expression we can use the `${expression}` notation for that.

```
var name = 'Rohit Jindal';
var text = `My name is ${name}`;
console.log(text); // My name is Rohit Jindal
```

Multi-line strings means that you no longer have to use `\n` for new lines anymore.

Example:

Output:

```
Hello Rohit!
How are you?
```

edited Dec 1 '16 at 6:42

Dev
242 6 15

answered Nov 11 '16 at 8:56

Rohit Jindal
8,874 9 44 80

12

Backticks enclose template literals, previously known as template strings. Template literals are string literals that allow embedded expressions and string interpolation features.

Template literals have expressions embedded in placeholders, denoted by the dollar sign and curly brackets around an expression, i.e. `${expression}` . The placeholder / expressions get passed to a function. The default function just concatenates the string.

To escape a backtick, put a backslash before it:

```
`\`` === '`'; => true
```

Use backticks to more easily write multi-line string:

```
console.log(`string text line 1
string text line 2`);
```

or

```
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
```

vs. vanilla JavaScript:

```
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
```

Escape sequences:

- Unicode escapes started by `\u` , for example `\u00A9`
- Unicode code point escapes indicated by `\u{}` , for example `\u{2F804}`
- Hexadecimal escapes started by `\x` , for example `\xA9`
- Octal literal escapes started by `\` and (a) digit(s), for example `\251`

edited Oct 5 '18 at 17:57

Peter Mortensen
14.2k 19 88 115

answered Sep 8 '17 at 20:34

mrmaclean89
356 5 4

2

Summary:

Backticks in JavaScript is a feature which is introduced in ECMAScript 6 // ECMAScript 2015 for making easy dynamic strings. This ECMAScript 6 feature is also named **template string literal**. It offers the following advantages when compared to normal strings:

- In Template strings linebreaks are allowed and thus can be multiline. Normal string literals (declared with `''` or `""`) are not allowed to have linebreaks.
- We can easily interpolate variable values to the string with the `${myVariable}` syntax.

Example:

```
const name = 'Willem';
const age = 26;

const story = `
  My name is: ${name}
  And I'm: ${age} years old
```

Run code snippet Expand snippet

Browser compatibility:

Template string literal are natively supported by all major browser vendors (except Internet Explorer). So it is pretty save to use in your production code. A more detailed list of the browser compatibilities can be found here.

edited Oct 5 '18 at 17:54
Peter Mortensen
14.2k 19 88 115

answered Oct 2 '18 at 16:49
Willem van der Veen
6,607 4 41 48

2

Good part is we can make basic Maths directly:

```
let nuts = 7

more.innerHTML = `

<h2>You collected ${nuts} nuts so far!

<hr>

Double it, get ${nuts + nuts} nuts!!

`


<div id="more"></div>
```

Run code snippet Expand snippet

It became really useful in a factory function:

```
    Triple nuts: ${3 * it} <br>
    Your nuts encoded in BASE64:<br> ${btoa(it)}
  `
}

nut.oninput = (function(){
  out.innerHTML = nuts(nut.value)
})


<h3>NUTS CALCULATOR
<input type="number" id="nut">

<div id="out"></div>
```

Run code snippet Expand snippet

edited Jan 22 at 1:13

answered Jan 22 at 0:49

Cryptopat
1,991 1 20 24

1 did nobody else chuckle cmon now – looch Jun 3 at 13:06