


Template literals (Template strings)

Template literals are literals delimited with backtick (```) characters, allowing for [multi-line strings](#), [string interpolation](#) with embedded expressions, and special constructs called [tagged templates](#).

Template literals are sometimes informally called *template strings*, because they are used most commonly for [string interpolation](#) (to create strings by doing substitution of placeholders). However, a tagged template literal may not result in a string; it can be used with a custom [tag function](#) to perform whatever operations you want on the different parts of the template literal.

Syntax

```
JS 

`string text`

`string text line 1
  string text line 2`

`string text ${expression} string text`

tagFunction`string text ${expression} string text`
```

Parameters

string text

The string text that will become part of the template literal. Almost all characters are allowed literally, including [line breaks](#) and other [whitespace characters](#). However, invalid escape sequences will cause a syntax error, unless a [tag function](#) is used.

expression

An expression to be inserted in the current position, whose value is converted to a string or passed to `tagFunction`.

tagFunction

If specified, it will be called with the template strings array and substitution expressions, and the return value becomes the value of the template literal. See [tagged templates](#).


Description

Template literals are enclosed by backtick (```) characters instead of double or single quotes.


Along with having normal strings, template literals can also contain other parts called *placeholders*, which are embedded expressions delimited by a dollar sign and curly braces: `${expression}`. The strings and placeholders get passed to a function — either a default function, or a function you supply. The default function (when you don't supply your own) just performs [string interpolation](#) to do substitution of the placeholders and then concatenate the parts into a single string.

To supply a function of your own, precede the template literal with a function name; the result is called a [tagged template](#). In that case, the template literal is passed to your tag function, where you can then perform whatever operations you want on the different parts of the template literal.

To escape a backtick in a template literal, put a backslash (`\`) before the backtick.

```
JS   
  
`\` === "`"; // true
```


Dollar signs can be escaped as well to prevent interpolation.

```
JS   
  
`\${1}` === "${1}"; // true
```

Multi-line strings

Any newline characters inserted in the source are part of the template literal.

Using normal strings, you would have to use the following syntax in order to get multi-line strings:

```
JS   
  
console.log("string text line 1\n" + "string text line 2");  
// "string text line 1  
// string text line 2"
```

Using template literals, you can do the same with this:

JS

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

String interpolation

Without template literals, when you want to combine output from expressions with strings, you'd [concatenate them](#) using the [addition operator](#) `+`:

JS

```
const a = 5;
const b = 10;
console.log("Fifteen is " + (a + b) + " and\nnot " + (2 * a + b) + ".");
// "Fifteen is 15 and
// not 20."
```

That can be hard to read – especially when you have multiple expressions.

With template literals, you can avoid the concatenation operator — and improve the readability of your code — by using placeholders of the form `${expression}` to perform substitutions for embedded expressions:

JS


```
const a = 5;
const b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```

Note that there's a mild difference between the two syntaxes. Template literals [coerce their expressions directly to strings](#), while addition coerces its operands to primitives first. For more information, see the reference page for the [`+` operator](#).

Nesting templates

In certain cases, nesting a template is the easiest (and perhaps more readable) way to have configurable strings. Within a backtick-delimited template, it is simple to allow inner backticks by using them inside an `${expression}` placeholder within the template.


For example, without template literals, if you wanted to return a certain value based on a particular condition, you could do something like the following:

```
JS   
  
let classes = "header";  
classes += isLargeScreen()  
  ? ""  
  : item.isCollapsed  
    ? " icon-expander"  
    : " icon-collapser";
```

With a template literal but without nesting, you could do this:

```
JS   
  
const classes = `header ${  
  isLargeScreen() ? "" : item.isCollapsed ? "icon-expander" : "icon-collapser"  
}`;
```

With nesting of template literals, you can do this:

```
JS   
  
const classes = `header ${  
  isLargeScreen() ? "" : `icon-${item.isCollapsed ? "expander" : "collapser"}`  
}`;
```


Tagged templates

A more advanced form of template literals are *tagged* templates.

Tags allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions.

The tag function can then perform whatever operations on these arguments you wish, and return the manipulated string. (Alternatively, it can return something completely different, as described in one of the following examples.)

The name of the function used for the tag can be whatever you want.

```
JS   
  
const person = "Mike";  
const age = 28;  
  
function myTag(strings, personExp, ageExp) {  
  const str0 = strings[0]; // "That "  
  const str1 = strings[1]; // " is a "  
  const str2 = strings[2]; // "."  
  
  const ageStr = ageExp < 100 ? "youngster" : "centenarian";  
  
  // We can even return a string built using a template literal  
  return `${str0}${personExp}${str1}${ageStr}${str2}`;  
}  
  
const output = myTag`That ${person} is a ${age}.`;  
  
console.log(output);  
// That Mike is a youngster.
```


The tag does not have to be a plain identifier. You can use any expression with [precedence](#) greater than 16, which includes [property access](#), function call, [new expression](#), or even another tagged template literal.

```
JS 

console.log`Hello`; // [ 'Hello' ]
console.log.bind(1, 2)`Hello`; // 2 [ 'Hello' ]
new Function("console.log(arguments)")`Hello`; // [Arguments] { '0': [ 'Hello' ] }



function recursive(strings, ...values) {
  console.log(strings, values);
  return recursive;
}
recursive`Hello`World`;
// [ 'Hello' ] []
// [ 'World' ] []
```

While technically permitted by the syntax, *untagged* template literals are strings and will throw a [TypeError](#) when chained.

```
JS 

console.log(`Hello`World`); // TypeError: "Hello" is not a function
```

The only exception is optional chaining, which will throw a syntax error.

```
JS  

console.log?.`Hello`; // SyntaxError: Invalid tagged template on optional chain
console?.log`Hello`; // SyntaxError: Invalid tagged template on optional chain
```

Note that these two expressions are still parsable. This means they would not be subject to [automatic semicolon insertion](#), which will only insert semicolons to fix code that's otherwise unparsable.

```
JS  

// Still a syntax error
const a = console?.log
`Hello`
```

Tag functions don't even need to return a string!

```
function template(strings, ...keys) {
  return (...values) => {
    const dict = values[values.length - 1] || {};
    const result = [strings[0]];
    keys.forEach((key, i) => {
      const value = Number.isInteger(key) ? values[key] : dict[key];
      result.push(value, strings[i + 1]);
    });
    return result.join("");
  };
}

const t1Closure = template`${0}${1}${0}!`;
// const t1Closure = template(["", "", "", "!"], 0, 1, 0);
t1Closure("Y", "A"); // "YAY!"

const t2Closure = template`${0} ${"foo"}!`;
// const t2Closure = template(["", " ", "!"], 0, "foo");
t2Closure("Hello", { foo: "World" }); // "Hello World!"

const t3Closure = template`I'm ${"name"}. I'm almost ${"age"} years old.`;
// const t3Closure = template(["I'm ", ". I'm almost ", " years old."], "name", "age");
t3Closure("foo", { name: "MDN", age: 30 }); // "I'm MDN. I'm almost 30 years old."
t3Closure({ name: "MDN", age: 30 }); // "I'm MDN. I'm almost 30 years old."
```

The first argument received by the tag function is an array of strings. For any template literal, its length is equal to the number of substitutions (occurrences of `${...}`) plus one, and is therefore always non-empty.

For any particular tagged template literal expression, the tag function will always be called with the exact same literal array, no matter how many times the literal is evaluated.

JS



```
const callHistory = [];  
  
function tag(strings, ...values) {  
  callHistory.push(strings);  
  // Return a freshly made object  
  return {};  
}  
  
function evaluateLiteral() {  
  return tag`Hello, ${"world"}!`;  
}  
  
console.log(evaluateLiteral() === evaluateLiteral()); // false; each time `tag` is called, it  
returns a new object  
console.log(callHistory[0] === callHistory[1]); // true; all evaluations of the same tagged  
literal would pass in the same strings array
```

This allows the tag to cache the result based on the identity of its first argument. To further ensure the array value's stability, the first argument and its `raw` property are both [frozen](#), so you can't mutate them in any way.

Raw strings

The special `raw` property, available on the first argument to the tag function, allows you to access the raw strings as they were entered, without processing [escape sequences](#).

JS



```
function tag(strings) {  
  console.log(strings.raw[0]);  
}  
  
tag`string text line 1 \n string text line 2`;  
// Logs "string text line 1 \n string text line 2" ,  
// including the two characters '\' and 'n'
```

In addition, the `String.raw()` method exists to create raw strings just like the default template function and string concatenation would create.

JS



```
const str = String.raw`Hi\n${2 + 3}!`;
// "Hi\\n5!"

str.length;
// 6

Array.from(str).join(",");
// "H,i,\\,n,5,!"
```

`String.raw` functions like an "identity" tag if the literal doesn't contain any escape sequences. In case you want an actual identity tag that always works as if the literal is untagged, you can make a custom function that passes the "cooked" (i.e. escape sequences are processed) literal array to `String.raw`, pretending they are raw strings.

JS



```
const identity = (strings, ...values) =>
  String.raw({ raw: strings }, ...values);
console.log(identity`Hi\n${2 + 3}!`);
// Hi
// 5!
```

This is useful for many tools which give special treatment to literals tagged by a particular name.

JS



```
const html = (strings, ...values) => String.raw({ raw: strings }, ...values);
// Some formatters will format this literal's content as HTML
const doc = html`<!doctype html>
  <html lang="en-US">
    <head>
      <title>Hello</title>
    </head>
    <body>
      <h1>Hello world!</h1>
    </body>
  </html>`;
```

Tagged templates and escape sequences

In normal template literals, [the escape sequences in string literals](#) are all allowed. Any other non-well-formed escape sequence is a syntax error. This includes:

- `\` followed by any decimal digit other than `0`, or `\0` followed by a decimal digit; for example `\9` and `\07` (which is a [deprecated syntax](#))
- `\x` followed by fewer than two hex digits (including none); for example `\xz`
- `\u` not followed by `{` and followed by fewer than four hex digits (including none); for example `\uz`
- `\u{}` enclosing an invalid Unicode code point — it contains a non-hex digit, or its value is greater than `10FFFF`; for example `\u{110000}` and `\u{z}`

Note: `\` followed by other characters, while they may be useless since nothing is escaped, are not syntax errors.

However, this is problematic for tagged templates, which, in addition to the "cooked" literal, also have access to the raw literals (escape sequences are preserved as-is).

Tagged templates should allow the embedding of languages (for example [DSLs](#), or [LaTeX](#)), where other escapes sequences are common. Therefore, the syntax restriction of well-formed escape sequences is removed from tagged templates.

JS

```
latex`\unicode`;  
// Throws in older ECMAScript versions (ES2016 and earlier)  
// SyntaxError: malformed Unicode character escape sequence
```

However, illegal escape sequences must still be represented in the "cooked" representation. They will show up as `undefined` element in the "cooked" array:

JS

```
function latex(str) {  
  return { cooked: str[0], raw: str.raw[0] };  
}  
  
latex`\unicode`;  
  
// { cooked: undefined, raw: "\\unicode" }
```

Note that the escape-sequence restriction is only dropped from *tagged* templates, but not from *untagged* template literals:

JS

```
const bad = `bad escape sequence: \unicode`;
```