Template literals (Template strings)

Template literals are string literals allowing embedded expressions. You can use multi-line strings and string interpolation features with them.

They were called "template strings" in prior editions of the ES2015 specification.

Syntax

```
`string text`

`string text line 1
 string text line 2`

`string text ${expression} string text`

tag`string text ${expression} string text`
```

Description

Template literals are enclosed by the backtick (` `) (grave accent) character instead of double or single quotes.

Template literals can contain placeholders. These are indicated by the dollar sign and curly braces (`${expression}`). The expressions in the placeholders and the text between the backticks (` `) get passed to a function.

The default function just concatenates the parts into a single string. If there is an expression preceding the template literal (`tag` here), this is called a *tagged template*. In that case, the tag expression (usually a function) gets called with the template literal, which you can then manipulate before outputting.

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

```
`\`` === '`' // --> true
```

Multi-line strings

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

```
// "string text line 1
// string text line 2"
```

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

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

Expression interpolation

In order to embed expressions within normal strings, you would use the following syntax:

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

Now, with template literals, you are able to make use of the syntactic sugar, making substitutions like this more readable:

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

Nesting templates

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

For instance, if condition a is `true`, then `return` this templated literal.

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

In ES2015 with nested template literals:

```
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.

```
let person = 'Mike';
let age = 28;

function myTag(strings, personExp, ageExp) {
  let str0 = strings[0]; // "That "
  let str1 = strings[1]; // " is a "
  let str2 = strings[2]; // "."

  let ageStr;
  if (ageExp > 99){
```

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

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

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

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

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

Raw strings

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

Tagged templates and escape sequences

ES2016 behavior

As of ECMAScript 2016, tagged templates conform to the rules of the following 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 "`\0o`" and followed by one or more digits, for example `\0o251`

This means that a tagged template like the following is problematic, because, per ECMAScript grammar, a parser looks for valid Unicode escape sequences, but finds malformed syntax:

```
latex`\unicode`
// Throws in older ECMAScript versions (ES2016 and earlier)
```

Specifications

Specification
ECMAScript Language Specification (ECMAScript) # sec-template-literals

Browser compatibility

Report problems with this compatibility data on GitHub

Template literals	
Chrome	41
Edge	12

WebView Android	62
Chrome Android	62
Firefox for Android	53
Opera Android	46
Safari on iOS	11
Samsung Internet	8.0
Node.js	8.10.0