JavaScript Template Literals In Depth

Summary: in this tutorial, you will learn about JavaScript template literal, which makes working with a string template easier.

Before ES6, you use single quotes (`'`) or double quotes (`"`) to wrap a string literal, which has very limited functionality.

To address more complex problems, ES6 introduced template literals, providing a safer and cleaner way to work with strings.

In ES6, you create a template literal by wrapping your text in backticks (`) as follows:

```
let simple = `This is a template literal`;
```

By doing this, you can get the following features:

- **A multiline string**: a string that can span multiple lines.
- **String formatting**: substitute parts of a string with the values of variables or expressions. This feature is also known as *string interpolation*.
- **HTML escaping**: the ability to transform a string to make it safe for inclusion in HTML.

The basic syntax of JavaScript template literals

As mentioned earlier, instead of using single quotes or double quotes, a template literal uses backticks, as shown in the following example:

```
let str = `Template literal in ES6`;

console.log(str);// Template literal in ES6
```

```
console.log(str.length); // 23
console.log(typeof str);// string
```

Using the backticks, you can freely use the single or double quotes in the template literals without escaping.

```
let anotherStr = `Here's a template literal`;
```

If a string contains a backtick, you must escape it using a backslash (`\`) :

```
let strWithBacktick = `Template literals use backticks \` insead of quotes`;
```

Multiline strings

Before ES6, you use the following technique to create a multi-line string by manually including the newline character (`\n`) in the string as follows:

```
let msg = 'Multiline \n\
string';

console.log(msg);
//Multiline
//string
```

Note that the backslash (`\`) placed after the newline character (`\n`) indicates the continuation of the string rather than a new line.

This technique, however, is not consistent across JavaScript engines. Therefore, it was pretty common to create a multiline string that relies on an array and string concatenation as follows:

```
let msg = ['This text',
         'can',
         'span multiple lines'].join('\n');
```

The template literals allow you to define multiline strings more easily because you need to add a new line in the string wherever you want:

```
let p =
`This text
can
span multiple lines`;
```

Note that the whitespace is a part of the string. Therefore, you need to ensure that the text lines up with proper indentation. Suppose you have a `post` object:

```
let post = {
    title: 'JavaScript Template Literals',
    excerpt: 'Introduction to JavaScript template literals in ES6',
    body: 'Content of the post will be here...',
    tags: ['es6', 'template literals', 'javascript']
};
```

The following code returns the HTML code of the `post` object. Note that we use the object destructuring technique to assign the properties of the `post` object to individual variables : `title` , `excerpt` , `body` , and `tags` .

```
let {title, excerpt, body, tags} = post;

let postHtml = `<article>
<header>
    <h1>${title}</h1>
</header>
<section>
    <div>${excerpt}</div>
    <div>${body}</div>
</section>
<footer>
    <ul>
      ${tags.map(tag => `<li>${tag}</li>`).join('\n      ')}
```

```
    </ul>
</footer>`;
```

The following is the output of the variable `postHtml`. Notice how we used the spacing to indent the `<li>` tags correctly.

```
<article>
  <header>
    <h1>JavaScript Template Literals</h1>
  </header>
  <section>
    <div>Introduction to JavaScript template literals in ES6</div>
    <div>Content of the post will be here...</div>
  </section>
  <footer>
    <ul>
      <li>es6</li>
      <li>template literals</li>
      <li>javascript</li>
    </ul>
  </footer>
</article>
```

Variable and expression substitutions

At this point, a template literal is essentially an improved version regular JavaScript string. The key difference is substitutions, which let you embed variables and expressions in a string.

The JavaScript engine automatically replaces these variables and expressions with their values, a feature known as string interpolation.

To instruct JavaScript to substitute a variable and expression, you place the variable and expression in a special block like this:

```
${variable_name}
```

For example:

```
let firstName = 'John',
    lastName = 'Doe';

let greeting = `Hi ${firstName}, ${lastName}`;
console.log(greeting); // Hi John, Doe
```

The substitution `${firstName}` and `${lastName}` access the variables `firstName` and `lastName` to insert their values into the `greeting` string.

The `greeting` variable then holds the result of the substitutions. The following example substitutes an expression instead:

```
let price = 8.99,
    tax = 0.1;

let netPrice = `Net Price:$${(price * (1 + tax)).toFixed(2)}`;

console.log(netPrice); // netPrice:$9.89
```

Tagged templates

A template tag carries the transformation on the template literal and returns the result string.

You place the tag at the beginning of the template before the backtick (`) character as follows:

```
let greeting = tag`Hi`;
```

In this example, `tag` is the template tag that applies to the `Hi` template literal. The `tag` can be any function with the following signature:

```
function tag(literals, ...substitutions) {
    // return a string
}
```

In this function:

- The `literals` parameter is an array that contains the literal strings.
- The `substitutions` parameter contains the subsequent arguments interpreted for each substitution.

See the following example:

```
function format(literals, ...substitutions) {
  let result = '';

  for (let i = 0; i < substitutions.length; i++) {
    result += literals[i];
    result += substitutions[i];
  }
  // add the last literal
  result += literals[literals.length - 1];
  return result;
}

let quantity = 9,
  priceEach = 8.99,
  result = format`${quantity} items cost $${(quantity * priceEach).toFixed(
    2
  )}.`;

console.log(result); // 9 items cost $80.91.
```

In this example, the `format()` function accepts three arguments: the `literals` array and two other arguments stored in the `substitutions` array.

The first argument is the `literals` array that contains three elements:

- An empty string before the first substitution (”). Note that the first argument of the literals array is an empty string.
- A string `'items cost'` that is located between the first and the second substitutions.
- A string that follows the second substitution (`'.'`)

The second argument is `9`, which is the interpreted value of the `quantity` variable. It becomes the first element of the `substitutions` array. The third argument is `80.91`, which is the interpreted value of the expression `(quantity * priceEach).toFixed(2)`. It becomes the second element of the substitution array.

Summary

- Use the backticks to create a string literal for string interpolation.