Template Literals

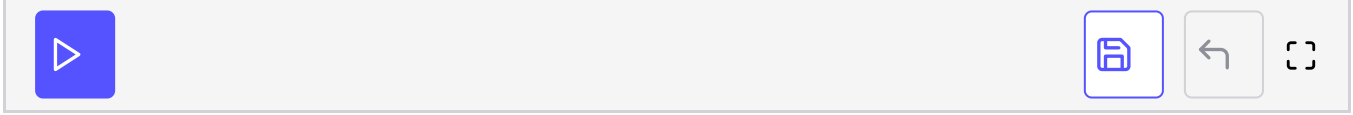
WE'LL COVER THE FOLLOWING

Template Literals

Introduced in ES6 is a new way in which we can create a string, and that is the Template Literal. With it comes new features that allow us more control over dynamic strings in our programs. Gone will be the days of long string concatenation!

To create a template literal, instead of `'` or `"` quotes we use the `` ` `` character. This will produce a new string, and we can use it in any way we want.

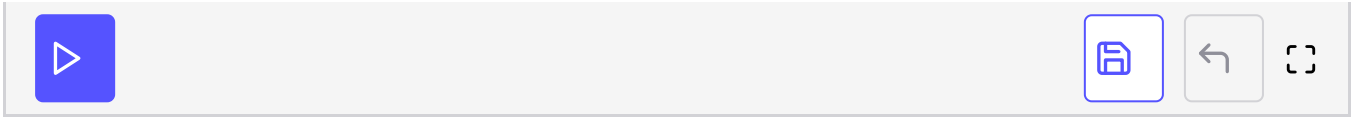
```
let newString = `A string`;
console.log(newString);
```



Multi-line

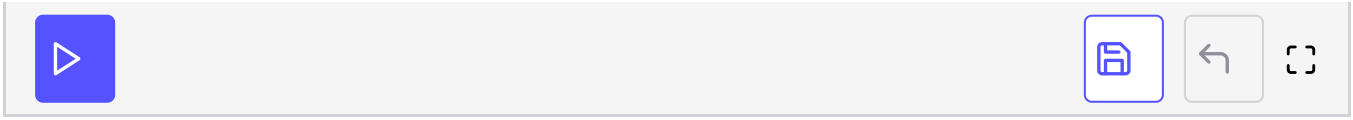
The great thing about Template Literals is that we can now create multi-line strings! In the past, if we wanted a string to be on multiple lines, we had to use the `\n` or new line character.

```
let myMultiString = 'Some text that I want\nOn two lines!';

console.log(myMultiString);
```



With a Template Literal string, we can just go ahead and add the new line into the string as we write it.

```
let myMultiString= `This will be
on two lines!`;

console.log(myMultiString);
```

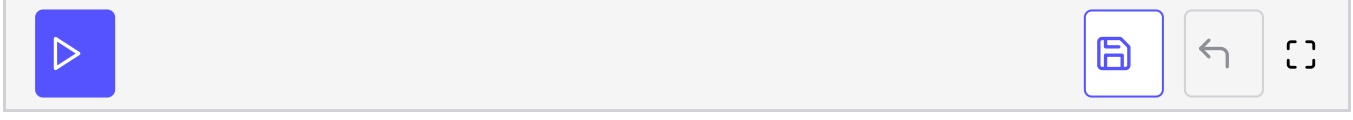


This will produce a string with a new line in it. The ability to do this with expressions makes Template Literals a really nice templating language for building of bits of HTML that we will cover later. But what about concatenation? Let's look at how we can dynamically add values into our new Template Literals.

Expressions

In the new Template Literal syntax we have what are called expressions, and they look like this: `${expression}`. Consider the code below.

```
let name = `Ryan`;

console.log(`Hi my name is ${name}`);
```



The `${}` syntax allows us to put an expression in it and it will produce the value, which in our case above is just a variable that holds a string! There is something to note here: if you wanted to add in values, like above, you do not need to use a Template Literal for the `name` variable. It could just be a regular

string.

```
console.log(`Hi my name is ${'Ryan'}`);
```

This will produce the same output. These expressions do more than just let us put in variables that contain strings in them. We can evaluate any sort of expressions that we would like.

```
let price = 19.99;
let tax = 1.13;

let total = `The total prices is ${price * tax}`;
console.log(total);
```

We can also use this with a more complex object.

```
let person = {
    firstName: `Ryan`,
    lastName: `Christiani`,
    sayName() {
        return `Hi my name is ${this.firstName} ${this.lastName}`;
    }
};

console.log(person.sayName());
```

Here we have a `person` object with a `sayName()` method on it: this uses the shorthand method definition we looked at in the Objects chapter! We can access the properties from an object inside of the `${}` syntax.

HTML Templates

With the ability to have multi-line strings and use Template Expressions to add content into our string, this makes it really nice to use for HMTL templates in our code.

Lets imagine that we get some data from an API that looks something like this:

```
{
    "id": 1,
    "name": "Bulbasaur",
    "base_experience": 64,
    "height": 7,
    "is_default": true,
    "order": 1,
    "weight": 69,
    ...
}
```

This “imaginary” API is of course the pokeapi! With this data structure in mind, let’s create the markup that would show this Pokemon.

```
function createMarkup(data) {
    return `
        <article class="pokemon">
            <h3>${data.name}</h3>
            <p>The Pokemon ${data.name} has a base experience of ${data.base_experience}, the
        </article>
    `
}
```

Without having to use a library like Handlebars or Mustache we can create nice and easy-to-use templates in our JavaScript!

Let’s look at our Pokeman example in action (*note: we are using jQuery to modify the DOM. You can ignore this example if you don’t understand jQuery*)

HTML JavaScript

javascript

```
let data = {
    "id": 1,
    "name": "Bulbasaur",
    "base_experience": 64,
    "height": 7,
    "is_default": true,
    "order": 1,
    "weight": 69
};

function createMarkup(data) {
    return `
        <article class="pokemon">
            <h3>${data.name}</h3>
            <p>The Pokemon ${data.name} has a base experience of ${data.base_experience}
        </article>
    `
}

$(document).ready(function () {
$('#content').html(createMarkup(data));
```

```
});

```

output

Bulbasaur

The Pokemon Bulbasaur has a base experience of 64, they also weigh 69

Tagged Templates

Another features of Template Literals is the ability to create Tagged Template Literals. The way this works is that you create a function and this function will look like any other function, however when you execute it, that is when it looks different. Consider the function below.

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

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

First off, you will notice there are no `()` when we call the function! We apply a Template Literal where the parentheses would be. As a parameter to our function we get an array of the strings in our literal. Let's expand on the string we send to the function and we will have it include an expression, and we will include a new parameter in our function as well.

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

When we use an expression we can access that from the next parameters and this keeps going. Say we added another expression.

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

This is pretty powerful: it allows you to take the data used in a string and manipulate it to your liking.

Reusable Templates

Let's look at a simple use case for this. If you remember from above, we saw how Template Literals work really great for, well, making templates! Let's take that a step further and create a function that would allow us to create reusable templates. The idea here is that we can create the initial template and then pass in data for it to use later.

```
const student = {
    name: "Ryan Christiani",
    blogUrl: "http://ryanchristiani.com"
}

const studentTemplate = templater`<article>
    <h3>${'name'} is a student at HackerYou</h3>
    <p>You can find their work at ${'blogUrl'}.</p>

</article>`;

const myTemplate = studentTemplate(student);
console.log(myTemplate);
//Output will look like this!
//<article>
//    <h3>Ryan Christiani is a student at HackerYou</h3>
//    <p>You can find their work at http://ryanchristiani.com.</p>
//</article>
```

Let's look at implementing our `templater` function.

```
const templater = function(strings,...keys) {

}
```

The first thing you will notice is this `...keys` parameter. The `...` syntax is what's called Rest Parameters; it basically will gather any parameters the function has and create an array for us. We will dig deeper into that in the Spread Operator & Rest Parameters chapter.

The next thing we want to do is return a function that is going to access our object. The returning of the function is what allows us to call and pass in our student data, like this: `studentTemplate(student)`.

```
const templater = function(strings,...keys) {
    return function(data) {

    }
}
```

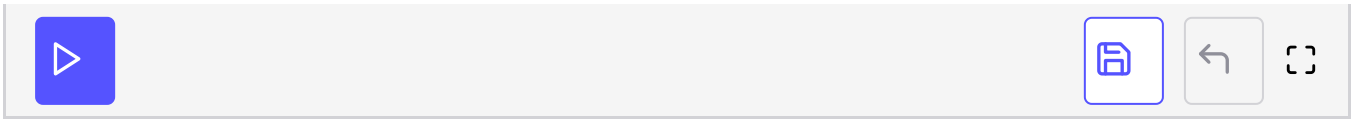
With this data now available to us we need to perform some manipulation. The process is as follows. First, we need to create a copy of the `strings` array. We make a copy in case we want to reference the original later. Then we need to loop through the array of `keys`, and for each one of those, grab the data from the object that matches the key (notice how in this example we pass in a string in the `${}`) and place it in our array where needed. Finally, we need to join it all back together as a string and return it from the function!

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

Let's see our example in action:

```
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 student = {
    name: "Ryan Christiani",
    blogUrl: "http://ryanchristiani.com"
}

const studentTemplate = templater`<article>
    <h3>${'name'} is a student at HackerYou</h3>
    <p>You can find their work at ${'blogUrl'}.</p>

</article>`;

const myTemplate = studentTemplate(student);
console.log(myTemplate);
//Output will look like this!
//<article>
//    <h3>Ryan Christiani is a student at HackerYou</h3>
//    <p>You can find their work at http://ryanchristiani.com.</p>
//</article>
```



You will notice that this is not an exhaustive example. We have no way to accommodate nested data or array values; it is simply just strings. But I hope this example helps to illustrate what you can start doing with Tagged Template Literals.