Apply, Call, Bind, & arguments

Learn 'apply', 'call', and 'bind', three functions that give us control over the 'this' value inside of a function. We'll see how these methods allow us to write functions that can accept any number of arguments.

In the next article, we're going to cover the full rules of `this`. First, we're going to showcase one more way to explicitly set the value of `this` in a function. We can use the `apply`, `call`, and `bind` methods. These are simple methods, but crucial to understanding how to read code and write proper object oriented JavaScript.

Occasionally, it's more useful and makes more sense for the `this` value to be some other object than what JavaScript wants to set it to. We can override some of the automatic cases of `this` inside functions by using these three methods.

Occasionally, we want to use a method of one object on another. Remember that in OOP, methods are meant to act on the object that they are a property of. It's often useful to take these methods and apply them (no pun intended) to different objects to perform the same action.

This prevents code duplication and makes it clear that we are using a method on a different object than it was intended.

call

`Function.call` allows us to set the `this` value of a function manually. Instead

`Function.call` allows us to set the `this` value of a function manually. Instead of simply calling a function like `fn()`, we use `fn.call(param)`, passing in the object we want `this` to equal as the parameter. This code block shows the output if we were running it in Chrome.

```
function logThis() {
    console.log(this);
}

const obj = { val: 'Hello!' };

logThis(); // -> Window {frames: Window, postMessage: ƒ, …}
logThis.call(obj); // -> { val: 'Hello!' };
```

We can see that the first function call works as expected, logging the global object. The second calls `logThis`, passing in `obj`, which gets set as the `this` value in that function call.

`call` also allows us to pass in parameters to the function being called. Anything given after the object to be bound to `this` will be passed along to the function.

```
function logThisAndArguments(arg1, arg2) {
    console.log(this);
    console.log(arg1);
    console.log(arg2);
}

const obj = { val: 'Hello!' };

logThisAndArguments('First arg', 'Second arg');
// -> Window {frames: Window, postMessage: ƒ, …}
// -> First arg
// -> Second arg

logThisAndArguments.call(obj, 'First arg', 'Second arg');
// -> { val: 'Hello!' }
// -> First arg
// -> Second arg
```

Using unknown function parameters

One particularly useful application of `call` is to work with an unknown set of parameters in a function. Let’s say we want to write a function that can take in as many numbers as a user wishes to provide, then add them up.

`arguments`

Inside every function, JavaScript gives us access to the `arguments` object. This is an *array-like* object that contains the arguments passed in to the function, indexed.

```
function add() {
    console.log(arguments);
}

add(4); // -> { '0': 4 }
add(4, 5); // -> { '0': 4, '1': 5 }
add(4, 5, 6); // -> { '0': 4, '1': 5, '2': 6 }
```

As we can see, `arguments` is an object with each of the property keys representing indices. It's not a true array but emulates one. Often, we want to turn this into a true array to be able to use native array methods such as `map`, `forEach`, etc. To do this, we can use `call` with `Array.slice`.

We use `Array.slice`, a pure function that returns a copy of an array if called with no arguments

`Array.slice` internally creates a copy of the original array by referencing `this`. By calling `Array.slice` on our `arguments` object, we are returned a new, true array, created from `arguments`.

```
function add() {
    const args = [].slice.call(arguments);
    console.log(args);
}

add(4, 5, 6); // -> [ 4, 5, 6 ]
```

Using this new `args` array, we can continue to write out our `add` function.

apply

`Function.apply` works the same exact way as `call`, except instead of passing in arguments one by one, we pass in an array of arguments that gets spread into the function.

```
function logThisAndArguments(arg1, arg2) {
    console.log(this);
    console.log(arg1);
    console.log(arg2);
}

const obj = { val: 'Hello!' };

logThisAndArguments('First arg', 'Second arg');
// -> Window {frames: Window, postMessage: ƒ, …}
// -> First arg
// -> Second arg

logThisAndArguments.apply(obj, ['First arg', 'Second arg']);
// -> { val: 'Hello!' }
// -> First arg
// -> Second arg
```

Note that the only difference when using `apply` is seen on line 14 above. The arguments to `logThisAndArguments.apply`, after `obj`, are inside an array.

bind

`Function.bind` works differently than the other two. Similarly to `call`, it takes in a `this` value and as many more parameters as we'd like to give it, and it'll pass those extra parameters to the function being called.

However, instead of calling the function immediately, `bind` *returns a new function*. This new function has the `this` value and the parameters already set and bound. When it's invoked, it'll be invoked with those items already in place.

```
function logThisAndArguments(arg1, arg2) {
    console.log(this);
    console.log(arg1);
    console.log(arg2);
}

const obj = { val: 'Hello!' };
const fnBound = logThisAndArguments.bind(obj, 'First arg', 'Second arg');

console.log(fnBound);
// -> [Function: bound logThisAndArguments]

fnBound();
// -> { val: 'Hello!' }
// -> First arg
// -> Second arg
```

Here they are all together.

```
function logThisAndArguments(arg1, arg2) {
    console.log(this);
    console.log(arg1);
    console.log(arg2);
}

const obj = { val: 'Hello!' };

// NORMAL FUNCTION CALL
logThisAndArguments('First arg', 'Second arg');
// -> Window {frames: Window, postMessage: ƒ, …}
// -> First arg
// -> Second arg

// USING CALL
logThisAndArguments.call(obj, 'First arg', 'Second arg');
// -> { val: 'Hello!' }
// -> First arg
// -> Second arg

// USING APPLY
logThisAndArguments.apply(obj, ['First arg', 'Second arg']);
// -> { val: 'Hello!' }
// -> First arg
// -> Second arg

// USING BIND
const fnBound = logThisAndArguments.bind(obj, 'First arg', 'Second arg');
fnBound();
// -> { val: 'Hello!' }
// -> First arg
// -> Second arg
```

These three functions are vital to JavaScript and you'll use them extensively in your career. You're also almost certain to be asked about them in a job interview.

In the next article, we'll find out exactly how the rules behind `this` work.

That's it.