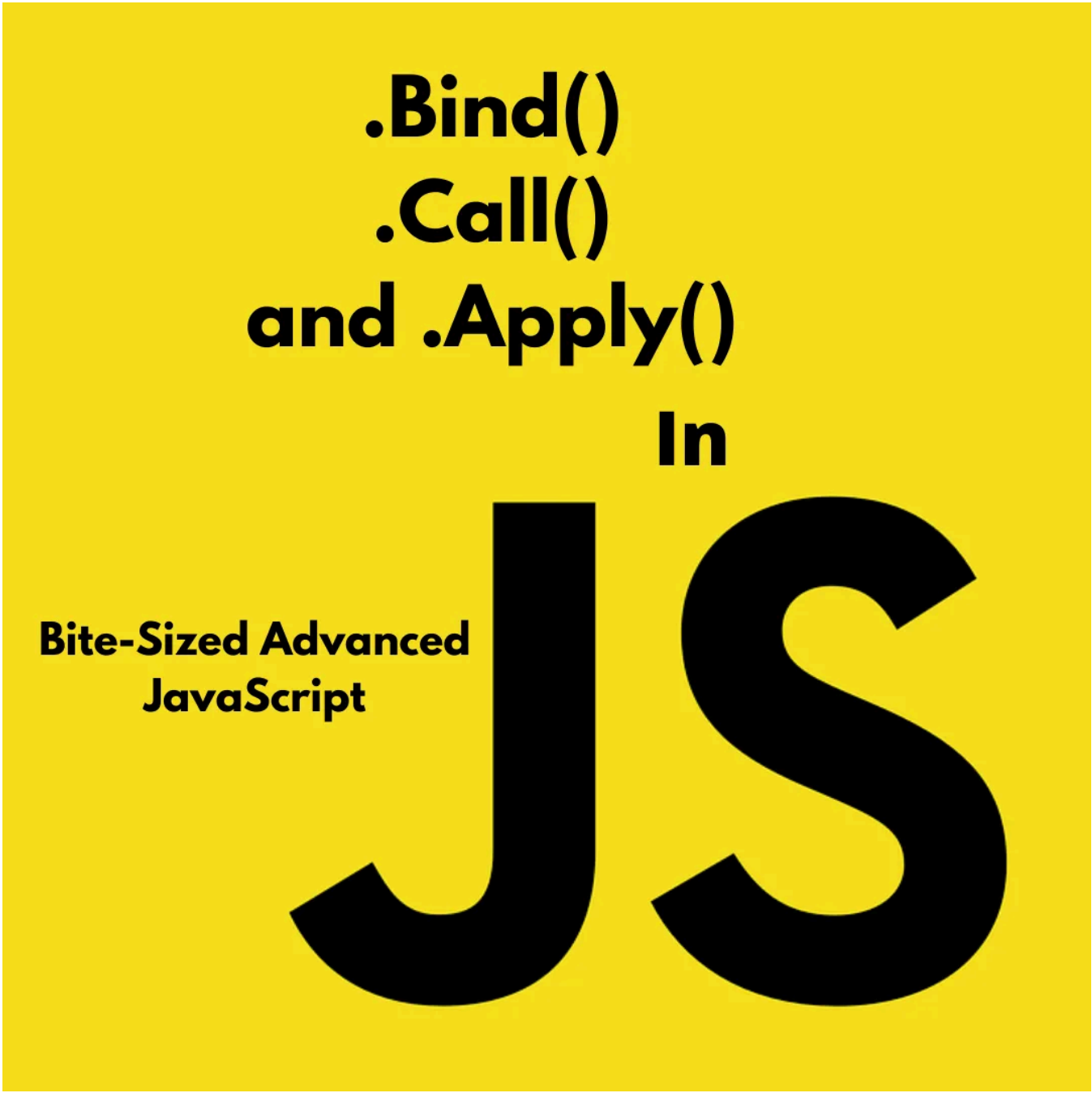
codeburst · Follow publication

JavaScript Under The Hood Pt. 4: Bind(), Call(), and Apply()

7 min read · Oct 5, 2020

Colton Kaiser Follow

Listen Share



When I first got into writing JavaScript, if there was anything that made me scratch my head the most it was the concepts of .bind(), .call(), and .apply(). The problem is that, as with many intermediate/advanced JavaScript concepts, they're often taught in a way that makes them seem more complicated than they need to be.

My goal here is that, by the end of this article, you'll no longer need to scratch your head at these three concepts and — in fact — be able to explain them to others.

What are bind(), call(), and apply()? How do they work?

To answer this question it's important to first remember that, **in JavaScript, all functions are objects. This means that they can have properties and methods, just like any other object.** Functions are a special kind of object in that they come with a variety of built-in properties (having a code property that is invocable, having an optional name, and the three methods call(), apply(), and bind()).

> Functions are a special type of object in that they come with a variety of built-in properties and methods, three of which are call(), apply(), and bind().

.bind()

Let's create a `person` object with two properties and a method. The method will be a `getFullName()` function expression that uses the 'this' keyword.

Important side note: In a method (a function property in an object), the 'this' keyword actually points to the object it is contained in, NOT the function itself (this is a weird and sometimes frustrating part of JavaScript that can result in hard-to-catch bugs — but not for you.)

```
var person = {
  firstname: 'John',
  lastname: 'Doe',
  getFullName: function() {

    var fullname = this.firstname + ' ' + this.lastname
    return fullname

  }
}
```

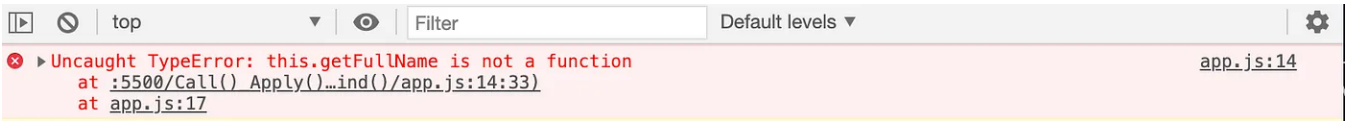
Person Object.js hosted with ❤ by GitHub view raw

Now let's create a `logName()` function that exists outside our `person` object.

```
//This will fail, because right now 'this' is pointing to the global object,
//which doesn't have a getFullName() function attached to it.
var logName = function(lang1, lang2) {

  console.log('Logged: ' + this.getFullName())
}

logName()
//Uncaught TypeError: this.getFullName is not a function
```

Bind() example pt. 1.js hosted with ❤ by GitHub view raw

If we call `logName()` right now, we get an error. This is because `logName()` is defined in the global scope, so 'this' is pointing at the global object, which does not have a `getFullname()` method. If you're not familiar with the global object and/or global environment, I'd encourage you to look over my post on the subject here.



Wouldn't it be nice if we could control what the 'this' keyword points to?

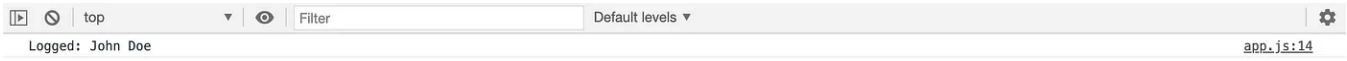
Enter bind().

Let's create a new function called `logPersonName()` that uses the built-in .bind() property on our `logName` function object. **Whatever object we pass as an argument to .bind() becomes what the 'this' variable points to when the function is run.**

```
var logName = function(lang1, lang2) {

  console.log('Logged: ' + this.getFullName())
}

var logPersonName = logName.bind(person)
logPersonName()
//We pass the bind() function whatever we want to be the "this" variable when the funct

//The reason we're not doing logName().bind is because we're simply using the function
//an object and accessing its built-in bind() property.

//The bind() function returns a copy of logName and sets it up so whenever it is run,
//it has person as the 'this' variable.
```

Bind() example pt. 2.js hosted with ❤ by GitHub view raw

Oh, look! Now our ‘this’ variable is pointing to the `person` object, accessing its properties and working exactly as expected.



Notice that we aren’t invoking the function before we call `.bind()`. Remember what we said at the beginning. Functions are objects, which means that they don't need to be immediately invoked. **We're using the function as an object and accessing its built-in bind() property.** This is why we're writing `logName.bind()` and **not** `logName().bind()`. The latter would be **invoking** the function and trying to access a `.bind()` method on the return value, which it obviously does not have.

> Whatever object we pass as an argument to .bind() becomes what the ‘this’ variable points to when the function is ran.

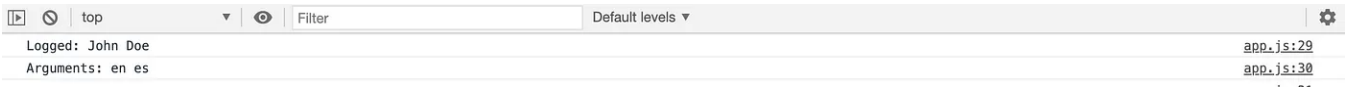
The bind() function returns a new function. It makes a copy of `logName()` and tells the JavaScript engine: "Whenever this copy of `logName()` is invoked, set its 'this' keyword to reference `person` during its execution context". This behavior of **returning a new function/making a copy** is different than call() and apply(), which we will get to momentarily.

Now that we have a copy of our `logName()` function referencing the `person` object (`logPersonName()`), **let's utilize its arguments.**

```
var person = {
  firstname: 'John',
  lastname: 'Doe',
  getFullName: function() {

    var fullname = this.firstname + ' ' + this.lastname
    return fullname

  }
}

var logName = function(lang1, lang2) {

  console.log('Logged: ' + this.getFullName())
  console.log('Arguments: ' + lang1 + ' ' + lang2)
}

var logPersonName = logName.bind(person)
logPersonName('en', 'es')
//Logged: John Doe
//Arguments: en es
```

Bind() example pt. 3.js hosted with ❤️ by GitHub view raw

Now we get:



One last quick note about bind(): We can only call bind() on a function once. Trying to use it subsequently will not do anything.

Example (Thanks to Krusader for providing this example):

```
function printGreetings () {
    return console.log(this.phrase);
};

const dataGreetings = {
    phrase: "Hello World!",
};

const dataFarewell = {
    phrase: "Goodbye for now!",
};
```

```
// first attempt to bind printGreetings to dataGreetings
const printHello = printGreetings.bind(dataGreetings);

printHello();
// -> "Hello World!"

// second attempt won't change anything
const printFarewell = printHello.bind(dataFarewell);

printFarewell();
// -> can't be bound twice so will print: "Hello World!"
```

.call()

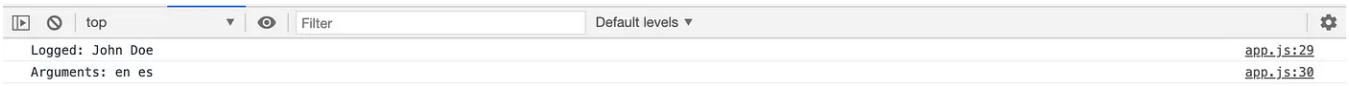
call() is similar to bind() in that it allows you to pass in what you'd like a 'this' value to point to. Except **call() doesn't make a copy. It invokes the function immediately.** Allowing you to pass in the 'this' reference and any arguments while calling the function simultaneously.

Let's try calling `logName()` by using call(). We'll let it know to reference `person` as its 'this' variable and give it two languages as arguments.

```
var person = {
  firstname: 'John',
  lastname: 'Doe',
  getFullName: function() {

    var fullname = this.firstname + ' ' + this.lastname
    return fullname

  }
}

var logName = function(lang1, lang2) {

  console.log('Logged: ' + this.getFullName())
  console.log('Arguments: ' + lang1 + ' ' + lang2)
}

logName.call(person, 'en', 'es')
//Logged: John Doe
//Arguments: en es

//.call() is the same thing as invoking a function "()".
//Difference is you can pass in what you want the 'this' variable to be at the same tim
//It is similar to .bind() except it calls the function immediately.
```

Call().js hosted with ❤️ by GitHub view raw

Here's what we're left with again:



Look similar? While it is similar, it's **not the same at all.** We're given the same return value as with bind() above, except this time we didn't make a copy and call the function immediately.

> Unlike bind(), call() doesn't make a copy of a function. It allows you to pass in the 'this' reference and any arguments, and then immediately invokes the function.

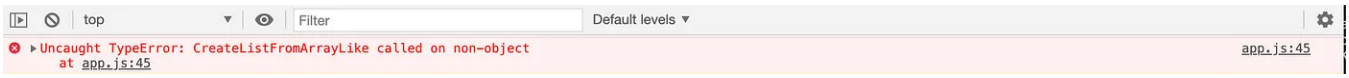
.apply()

This one's easy. It does the **exact same thing as call(). Except for one difference.**

Let's try replacing call() with apply() and see what happens.

```
logName.apply(person, 'en', 'es')
//Uncaught TypeError: CreateListFromArrayLike called on non-object
```

Apply() pt. 1.js hosted with ❤ by GitHub view raw



Hmm. Strange error. But it makes sense. **Apply() does the exact same thing as call(), but it requires the original function's arguments to be passed in as an array.**

Here's how it *should* look:

```
logName.apply(person, 'en', 'es')
//Uncaught TypeError: CreateListFromArrayLike called on non-object

logName.apply(person, ['en', 'es'])
//Logged: John Doe
//Arguments: en es
```

Apply() pt. 2.js hosted with ❤ by GitHub view raw

> Apply() does the exact same thing as call(), but it requires the original function's arguments to be passed in as an array.

So why would you ever use apply()? Well, it depends on how you'd like your data to be shown. Having the arguments as an array can be useful for mathematical situations, or simply for making the separation between the 'this' reference and the original function arguments easier to distinguish.

When would I ever use this?

A couple main uses for bind(), call(), and apply() are **function borrowing and function currying**.

Function borrowing example:

Function borrowing involves 'borrowing' a method from an object and applying it to another object. Can you think of how call() or apply() might be useful for this?

```
person.getFullName.apply(person2).
```

We've accessed our `getFullName()` method from `person` and then used apply() to specify that we'd like the 'this' variable to point to a `person2` object who will most likely have different first and last name properties (although the property names need to match up).

Function currying example:

Function currying creates a copy of a function with some preset parameters. Can you think of how bind() might be useful for this?

A simple example could be creating a function that multiplies two parameters. With function currying bind(), we can create copies of this function with two as a preset parameter.

```
function multiply(a, b) => {return a * b}

var multiplyByTwo = multiply.bind(this, 2)
```

Giving parameters to bind() sets permanent values to the passed in parameters when the copy is made. So in this case, we're setting the first parameter to always be 2. Ex: `multiplyByTwo(4)` would return `8`.

This concept can be very useful for mathematical situations that can benefit from some fundamental functions that can then be built on with other preset parameters.

In summary

- **Bind():** Returns a copy of a function, and takes an argument of what you'd like the 'this' variable to point to within the copied function. It can take additional arguments that act as permanent preset parameters for its copy of the original function.
- **Call():** Is similar to bind() in that it accepts the reference of the 'this' variable as an argument, but it calls the function immediately and does not create a copy.
- **Apply():** Works almost exactly the same as call(), except any arguments from the original function need to be passed as an array (this doesn't include the 'this' reference, which is outside the array).

When have you used .bind(), .call(), and .apply() in your programming? Let me know in the comments section.

If you liked this bite-sized piece of advanced JavaScript, feel free to check out the rest of the series!

JavaScript | Programming | Coding | Software Development | Software Engineering

Follow

Published in codeburst