Context in JavaScript

It's now time to dive into one of the most interesting concepts in JavaScript: the idea of **context**.

Programmers from the junior to senior level often confuse *scope* and *context* as the same thing - but that is not the case! Every function that is invoked has **both** a scope and a context associated with that function. *Scope* refers to the visibility and availability of variables, whereas *context* refers to the value of the `this` keyword when code is executed.

When you finish this reading you should be able to:

- Define a method that references `this` on an object
- Identify what `this` refers to in a code snippet
- Utilize the built in `Function#bind` to maintain the context of `this`

What about `this`?

When learning about objects we previously came across the idea of a *method*. A *method* is a function that is a value within an object and belongs to an object.

There will be times when you will have to know which object a method belongs to. The keyword `this` exists in every function and it evaluates to the object that is currently invoking that function. So the value of `this` relies entirely on **where** a function is invoked.

That may sound pretty abstract, so let's jump into an example:

```
let dog = {
  name: "Bowser",
  isSitting: true,

  stand: function () {
    this.isSitting = false;
    return this.isSitting;
  },
};

// Bowser starts out sitting
console.log(dog.isSitting); // prints `true`

// Let's make him stand
console.log(dog.stand()); // prints `false`

// He's actually standing now!
console.log(dog.isSitting); // prints `false`
```

Inside of a method, we can use the keyword `this` to refer to the object that is calling that method! So when calling `dog.stand()` and we invoke the code of the `stand` method, `this` will refer to the `dog` object.

Still skeptical? Don't take our word for it, check `this` (heh) out:

```
let dog = {
  name: "Bowser",

  test: function () {
    return this === dog;
  },
};

console.log(dog.test()); // prints `true`
```

In short, by using the `this` keyword inside a method, we can refer to values within that object.

Let's look at another example of this:

```
let cat = {
  purr: function () {
    console.log("meow");
  },
  purrMore: function () {
    this.purr();
  },
};

cat.purrMore();
```

Through the `this` variable, the `purrMore` method can access the object it was called on. In `purrMore`, we use `this` to access the `cat` object that has a `purr` method. In other words, inside of the `purrMore` function if we had tried to use `purr()` instead of `this.purr()` it would not work.

When we invoked the `purrMore` function using `cat.purrMore` we used a **method-style** invocation.

Method style invocations follow the format: `object.method(args)`. You've already been doing this using built in data type methods! (i.e. `Array#push`, `String#toUpperCase`, etc.)

Using *method-style invocation* (note the *dot notation*) ensures the method will be invoked and that the `this` within the method will be the object that method was called upon.

Now that we have gone over what `this` refers to - you can have a full understanding of the definition of context. **Context refers to the value of `this` within a function and `this` refers to where a function is invoked**.

Issues with scope and context

In the case of context the value of `this` is determined by *how* a function is invoked. In the above section we talked briefly about *method-style invocation*, where `this` is set to the object the method was called upon.

Let's now talk about what `this` is when using normal *function style invocation*.

If you run the following in Node:

```
function testMe() {
  console.log(this); //
}

testMe(); // Object [global] {global: [Circular], etc.}
```

When you run the above `testMe` function in Node you'll see that `this` is set to the `global` object. To reiterate: each function you invoke will have *both* a context and a scope. So even running functions in Node that are not defined explicitly on declared objects are run using the global object as their `this` and therefore their context.

When methods have an unexpected context

So let's now look at what happens when we try to invoke a method using an unintended context.

Say we have a function that will change the name of a dog object:

```
let dog = {
  name: "Bowser",
  changeName: function () {
```

```
    this.name = "Layla";
  },
};
```

Now say we wanted to take the `changeName` function above and call it somewhere else. Maybe we have a callback we'd like to pass it to or another object or something like that.

Let's take a look at what happens when we try to isolate and invoke just the `changeName` function:

```
let dog = {
  name: "Bowser",
  changeName: function () {
    this.name = "Layla";
  },
};

// note this is **not invoked** - we are assigning the function itself
let change = dog.changeName;
console.log(change()); // undefined

// our dog still has the same name
console.log(dog); // { name: 'Bowser', changeName: [Function: changeName] }

// instead of changing the dog we changed the global name!!!
console.log(this); // Object [global] {etc, etc, etc,  name: 'Layla'}
```

So in the above code notice how we stored the `dog.changeName` function *without invoking it* to the variable `change`. On the next line when we did invoke the `change` function we can see that we did not actually change the `dog` object like we intended to. We created a new key value pair for `name` on the global object! This is because we invoked change without the context of a specific object (like `dog`), so JavaScript used the only object available to it, the **global object**!

The above example might seem like an annoying inconvenience but let's take a look at what happens when calling something in the wrong context can be a big problem.

Using our `cat` object from before:

```
let cat = {
  purr: function () {
    console.log("meow");
  },
  purrMore: function () {
    this.purr();
  },
};

let notACat = cat.purrMore;
console.log(notACat()); // TypeError: this.purr is not a function
```

So in the above code snippet we attempted to call the `purrMore` function *without the correct Object for context*. Meaning we attempted to call the `purrMore` function on the global object! Since the global object does not have a `purr` method upon its `this` it raised an error. This is a common problem when invoking methods: invoking methods without their proper context.

Let's look at one more example of confusing `this` when using a callback. Incorrectly passing context is an inherent problem with callbacks. The `global.setTimeout()` method on the global object is a popular way of setting a function to run on a timer. The `global.setTimeout()` method accepts a callback and a number of milliseconds to wait before invoking the callback.

Let's look at a simple example:

```
let hello = function () {
  console.log("hello!");
};

// global. is a method of the global object!
global.setTimeout(hello, 5000); // waits 5 seconds then prints "hello!"
```

Expanding on the `global.setTimeout` method now using our `cat` from before let's say we wanted our `cat` to "meow" in 5 seconds instead of right now:

```
let cat = {
  purr: function () {
    console.log("meow");
  },
  purrMore: function () {
    this.purr();
  },
};

global.setTimeout(cat.purrMore, 5000); // 5 seconds later: TypeError: this.purr i
```

So what happened there? We called `cat.purrMore` so it should have the right context right? Nooooooope. This is because `cat.purrMore` is a callback in the above code! Meaning that when the `global.setTimeout` function attempts to call the `purrMore` function all it has reference to is the function itself. Since `setTimeout` is on the global object that means that the global object will be the context for attempting to invoke the `cat.purrMore` function.

Strictly protecting the global object

The accidental mutation of the global object when invoking functions in unintended contexts is one of the reasons JavaScript released "strict" mode in ECMAScript version 5. We won't dive too much into JavaScript's strict mode here, but it's important to know how strict mode can be used to protect the global object.

Writing and running code in strict mode is easy and much like writing code in "sloppy mode" (jargon for the normal JavaScript environment). We can run JavaScript in strict mode simply by adding the string "use strict" at the top of our file:

```
"use strict";

function hello() {
  return "Hello!";
}

console.log(hello); // prints "Hello!"
```

One of the differences of strict mode becomes apparent when trying to access the global object. As we mentioned previously, the global object is the context of invoked functions in Node that are not defined explicitly on declared objects.

So referencing `this` within a function using the global object as its context will give us access to the global object:

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

hello(); // Object [global] {etc, etc, etc }
```

However, strict mode will no longer allow you access to the global object in functions via the `this` keyword and will instead return `undefined`:

```
"use strict";
```

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

hello(); // undefined
```

Using strict mode can help us avoid scenarios where we accidentally would have mutated the global object. Let's take our example from earlier and try it in strict mode:

```
"use strict";

let dog = {
  name: "Bowser",
  changeName: function () {
    this.name = "Layla";
  },
};

// // note this is **not invoked** - we are assigning the function itself
let changeNameFunc = dog.changeName;

console.log(changeNameFunc()); // TypeError: Cannot set property 'name' of undefi
```

As you can see above, when we attempt to invoke the `changeNameFunc` an error is thrown because referencing `this` in strict mode will give us `undefined` instead of the global object. The above behavior is helpful for catching otherwise tricky bugs.

If you'd like to learn more about strict mode we recommend checking out the documentation.

Changing context using `bind`

Good thing JavaScript has something that can solve this problem for us: what is known as the **binding** of a context to a function.

From the Function.prototype.bind(), "The simplest use of `bind()` is to make a function that, no matter how it is called, is called with a particular `this` value".

Here is a preview of the syntax we use to `bind`:

```
let aboundFunc = func.bind(context);
```

So when we call `bind` we are returned what is called an exotic function. Which essentially means a function with it's `this` bound no matter where that function is invoked.

Let's take a look at example at `bind` in action:

```
let cat = {
  purr: function () {

    console.log("meow");
  },
  purrMore: function () {
    this.purr();
  },
};

let sayMeow = cat.purrMore;
console.log(sayMeow()); // TypeError: this.purr is not a function

// we can now use the built in Function.bind to ensure our context, our `this`,
// is the cat object
let boundCat = sayMeow.bind(cat);

// we still *need* to invoke the function
boundCat(); // prints "meow"
```

That is the magic of `Function#bind`! It allows you choose the context for your function. You don't need to restrict the context you'd like to bind to either - you can `bind`functions to any context.

Let's look at another example:

```
let cat = {
  name: "Meowser",
  sayName: function () {
    console.log(this.name);
  },
};

let dog = {
  name: "Fido",
};

let sayNameFunc = cat.sayName;

let sayHelloCat = sayNameFunc.bind(cat);
sayHelloCat(); // prints Meowser

let sayHelloDog = sayNameFunc.bind(dog);
sayHelloDog(); // prints Fido
```

Let's now revisit our above example of losing context in a callback and fix our context! Using the `global.setTimeout`function we want to call the`cat.purrMore`function with the context bound to the cat object.

Here we go:

```
let cat = {
  purr: function () {
    console.log("meow");
  },
  purrMore: function () {
    this.purr();
  },
};

// here we will bind the cat.purrMore function to the context of the cat object
const boundPurr = cat.purrMore.bind(cat);

global.setTimeout(boundPurr, 5000); // prints 5 seconds later: meow
```

Binding with arguments

So far we've talking of one of the the common uses of the `bind`function - binding a context to a function. However, bind will not only allow you to bind the context of a function but also to bind **arguments**to a function.

Here is the syntax for binding arguments to a function:

```
let aboundFunc = func.bind(context, arg1, arg2, etc...);
```

Following that train of logic let's look at example of binding arguments to a function, regardless of the context:

```
const sum = function (a, b) {
  return a + b;
};

// here we are creating a new function named add3
// this function will bind the value 3 for the first argument
const add3 = sum.bind(null, 3);

// now when we invoke our new add3 function it will add 3 to
// one incoming argument
console.log(add3(10));
```

Note that in the above snippet where we `bind` with `null` we don't actually use `this` in the `sum` function. However, since `bind` requires a first argument we can put in `null` as a place holder.

Above when we created the `add3` function we were creating a new bound function where the context was `null`, since the context won't matter, and the first argument will *always* be `3` for that function. Whenever we invoke the `add3` function all other arguments will be passed in normally.

Using `bind` like this gives you a lot of flexibility with your code. Allowing you to create independent functions that essentially do the same thing while keeping your code very DRY.

Here is another example:

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

const double = multiply.bind(null, 2);
const triple = multiply.bind(null, 3);

console.log(double(3)); // 6
console.log(triple(3)); // 9
```

What you learned

- How to define a method that references `this` on an object
- Identify what `this` refers to in a code snippet
- How to utilize the built in `Function#bind` to maintain the context of `this`