The Magic Of ***this***Keyword

JavaScript borrows **this**keyword from C++, where it is used to point to an instance of an object from within its own class definition. There is absolutely no confusion about this keyword in C++ because this keyword is never used for any other intent than to point to an instance of an object from its constructor!

The original designer of JavaScript decided to tie a *secondary*feature to **this**keyword. Not only is it used to ***point to an instance of an object from its own constructor or its methods****(hence,****this****)*it is also used to keep track of *execution context —*which is often based on where a function was *called*from.

This duality of **this**keyword is exactly what have wrecked havoc on learning and understanding how it works. It is also why **this**keyword can be so confusing even to programmers who come to JavaScript from C++.

**Overview**

So let’s summarize for JavaScript:

1. The **this**keyword is used to point to the *instance*of an object from its own constructor and methods (when used inside **function**or **class**scope.)
2. The **this**keyword also ***keeps track of execution context*** also often referred to *by some*as the *lexical scope* or lexical environment. Think of lexical scope as location in memory allocated for all local variables to that scope.
3. The link to the **execution context**can change: for example using **this**inside an *arrow function*is not the same as using this in *ES-style function*. Arrow functions are not constructors and cannot be used to instantiate an object. So they don’t even have their own **this**context. But they do have this keyword. So what does it point to? It is likely to be the parent context just outside of it. It has a “transparent” scope, I guess you can say.
4. The link to ***execution context***is also established when this is referred by a *callback*function, even if that function was defined inside an object’s constructor (when **function**or **class**is used as an object constructor.)

**Where It All Begins**

Let’s take a look at the original idea behind **this**keyword:

**function** Cat() {  
 **this**.name = "felix";  
 console.log(**this**); // Cat {name: "felix"}  
}**let** cat = **new** Cat(); // Cat {name: "felix"}

In this example function **Cat**is used as a constructor for the object of type **Cat**. No confusion whatsoever. The **this**referencepoints to itself. I mean… this is *why*it’s called **this**in the first place: “***this object***” we’re defining right now.

When used as a constructor function, the body of the function itself becomes the object construction area.

As expected, here **this**points to self: Cat { name: “felix” }

After instantiating **cat**, **cat.name**will be “felix”, because property name was attached to a future object instance **this**. The same goes for class keyword:

**class** Mouse {  
 constructor() {  
 **this**.name = "mappy";  
 console.log(**this**); // Mouse {name: "mappy"}  
 }  
}**let** mouse= **new** Mouse(); // Mouse{name: "mappy"}

Same thing is observed here. This makes perfect sense! If we just stopped here the **this**keyword in JavaScript would never cause any confusion whatsoever.

This is how it works in C++ too. And I think the original designer of JavaScript should have stopped here. But… instead the **this**keyword was also chosen to carry a link to *execution context*. This choice was beyond reason. Why not invent another keyword: let’s say **context**,and separate two distinct functionalities? However, now **this**is responsible for both features!

So how does this “other” **this**work?

**The Secondary Purpose Of this Keyword**

When used outside of *object instances*(previous examples) the **this**keyword takes on a completely new meaning.

***Sometimes a function is just a function:***

**function** abc() {  
 console.log(**this**);  
}abc(); // [**object Window**] -- points to window object

Here **this**points to [**object Window**] (or global scope.) Simply because it was called from the global scope (assuming we’re in <script> brackets.)

*But…*when you want to use the function as a constructor of an object, in other words, to create an object of type **abc**, the **this**keyword magically changes its context to the instance of the **type** object itself (no longer [**object Window**]):

**function** abc() {  
 console.log(**this**);  
}**let** type = **new** abc(); // abc{} -- self  
typeof type;

In other words object pointed to by **type**variable(self) becomes the context. This is why **this**is called **this**.

What about the previous example with window Object? How is that even **this** object anymore? Well, it isn’t.

It points to an object. Just not ***this*** object. In fact you could probably say it points to ***that*** object :-) And what ***that***object is will also change based on context from which a function was called.

Before *arrow functions*: **that** was exactly the name many programmers gave to the **this**object when used in another context.

First, let’s consider what the problem is:

**function** food(kind)

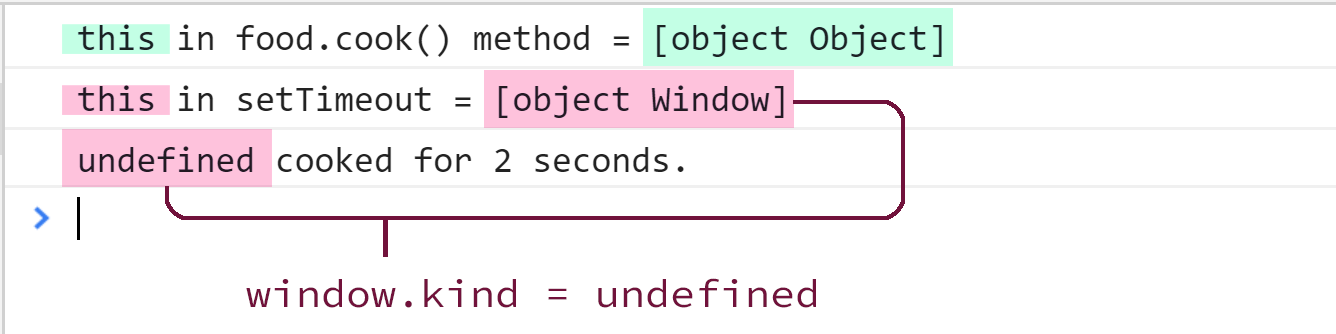
{ **this**.kind = kind;  
 **this**.cook = cook; // functions are hoisted, so it's perfectly  
 // fine to call or assign function names  
 // before they are defined.

**function** cook(sec) {  
 setTimeout(**function**() {  
 console.log(this.kind + " cooked for " + sec + " seconds.");  
 }, sec \* 1000);  
 }  
 }**let** soup = **new** food(**"soup"**); **// <--- this.kind = "soup"**

soup.cook(**2**);

**// undefined cooked for 2 seconds.**

Wait. What? I thought we assigned **this.kind = “soup”** when constructing the food object? Why does it say **undefined cooked for 2 seconds**?



When we called **setTimeout**function, it disconnected us from **this**keyword of the object. In callback of setTimeout**this**points to [**object Window**] and not the original **food** object aka [**object Object**].

In **setTimeout** we assumed we’re accessing **food.kind**property. That’s what anyone would do. Because the function is called from **food**object. So, naturally **this**and **this.kind**should refer to that object, right? Wrong.

But this is such an obvious case. In fact, it almost seems that **this***should*point to the object and not **window**. Well, that’s just how JavaScript works in this situation. So in the olden days, this problem was fixed as follows:

**function** food(kind)

{ **this**.kind = kind;  
 **this**.cook = cook;

**function** cook(sec) {

// remember, in cook(), this still points to food object  
 // and not the window object (like in setTimeout)!  
 // this means we can still capture it here, and pass it  
 // into setTimeout callback manually, so let's make a  
 // reference to **this** and call it **that**. (*or anything you want*)

**let** that = this;

// Now inside **setTimeout,** refer to **that.kind** not **this.kind:**  
 setTimeout(**function**() {  
 console.log(**that**.kind + " cooked for " + sec + " seconds.");  
 }, sec \* 1000);  
 }  
 }**let** soup = **new** food(**"soup"**);soup.cook(**2**); **// soup cooked for 2 seconds.**

Example with object:

**const** details = {  
 name: 'Arfat',  
 **friends: ['Bob', 'Alex'],**  
 **getFriends**: function () {  
 this.friends.forEach(**function** (friend) {  
 **console.log(this.name + " is friends with " + friend);**  
 });  
 }  
}details.getFriends();// undefined is friends with Bob  
// undefined is friends with Alex

Note the added **let that = this**; // *seems*like a *copy*of this object at first

No copies are created in **let that = this**assignment. In JavaScript we don’t make copies when assigning variables. *We create references*. From now on **that**variable name will *point*to the original **this**object.

Remember *functions*created using **function**keyword absorb all variables from the outside environment. So we can freely use **that**variable (that now points to **food**object) inside the callback function we passed to **setTimeout**.

What if this was **addEventListener**which also takes callback function? The same is true in this case. The problem with all callbacks inside functions is that they don’t link **this**keyword to the object from which they are called.

The whole reason **this**was still [**object Window**] and not **food**, was because at the end of the day the method **soup.cook(2)** was really called from global scope (same as [**object Window**])**.**This *secondary*function of **this**keyword to carry context took precedence.

That’s not what we needed. So we force-fed **that**inside the callback and made it point to the **food** object’s **this**.

The link is established. Now calling **soup.cook(2)** correctly outputs the message **“soup cooked for 2 seconds”**. Should you want to eat such a soup is up to you. Let me know how that goes. We’re just learning JavaScript here ;-)

**You hear a lot about “this binding” — but what in the world is it?**

Well, that’s exactly the idea we’ve just covered above.

You have just created your own **this**binding!

Inside *setTimeout***that**is now bound to **this**. To bind **this**is simply to wire it to some object. In some cases we need it to be [object Window] in other cases we might need it to point to a custom object.

Good news is that since ES6 we no longer *need to*do any of this by hand: there are better ways of doing this. Exactly how is explained in the next section.

**Arrow Functions To The Rescue**

Before arrow functions, programmers had to bind **this**to **that**manually.But arrow functions can automatically fix this problem. Arrow functions have a “transparent” scope. In other words in **setTimeout(()=>{this}, 1000)**the **this**keyword does not point to [object Window]. It points to whatever is outside of it. And in the previous case outside *is*the **food**object. This fixes the need to constantly have to bind **this**to the proverbial **that.**

And here is the modern version of previous code without **this/that**binding:

**function** food(kind) {  
 this.kind = kind;  
 this.cook = cook;  
 function cook(sec) {  
 setTimeout(**() => {**  
 console.log(this.kind + " cooked for " + sec + " seconds.");  
 }, sec \* 1000);  
 }  
 }  
 let soup = new food("soup");  
 soup.cook(2); // soup cooked for 2 seconds.

With object:

const details = {  
 name: 'Arfat',  
 friends: ['Bob', 'Alex'],  
 getFriends: function () {  
 **const that = this;**  
 this.friends.forEach(function (friend) {  
 console.log(**that.name** + " is friends with " + friend);  
 });  
 }  
}  
details.getFriends();// Arfat is friends with Bob  
// Arfat is friends with Alex

It works.

The **let that = this**nonsense is gone.

The **soup**and not *undefined*is cooked for 2 seconds!

For years programmers had to do this binding by hand. It’s annoying.

We replaced **function() {}** with arrow syntax **() => {}**

And that solved the whole problem. But at least we learned something new.

That’s what arrow functions were invented for. (*Although they also have other interesting properties: such as clean(er) code: when used together with Array’s higher-order functions.*)

**How to reconcile the two “this” use cases, without losing your mind?**

Well, **this** is the fun part.

There is a *sane*way of thinking about **this**in both contexts.

Remember how we saw that **this**points to ***instance***of an object in first case?

But in the second case it points to some other ***context***?

It all starts to make more sense if we recognize that **context**is***also***an **instance**of an object! Just not necessarily the one belonging to the scope we’re in. It is more like a link to the outside world. And the outside world is — roughly — the place from which the function was called.

In some cases JavaScript will decide for us whether to use the link to the outside world, or to use **this**as a reference to the instance of the object we are in the constructor’s scope of.

**Takeaway:** Context is always an instance to an object. Because when our JavaScript program begins, browser creates: **var window = new Window()**object behind the scenes. This is the very first “root” context created.

So when we are in **<script>** tags, we are actually inside the body of the Window constructor function. And this is why [**object Window**] is a prime candidate for being pointed to by **this**keyword in many cases. Because this is the object from which many functions are called.

Every function call will stem from that context. Calling a function inside a function will carry this context, creating a stack of execution contexts.

**So when does this \*not point to [object Window]?**

In the *second use case*when **this**points to the *instance of an execution context* rather than the *instance of the object*from constructor we’re in… what are the cases when this will not point to [**object Window**]?

That depends on whether the function was a callback, but also where the function that called the callback was called from.



Is this pure confusion or genius of JavaScript? I’ll let you decide.

**Keep writing code.**

These types of concepts take time to sink in.

In a lot of cases, you don’t even need to understand them in order to write good JavaScript code. But it’s probably in your interest to get to this level of detail because these types of questions are often asked on interviews.

Just write JavaScript and watch what happens. You will learn to understand **this**keyword by instinct the more you practice.

If you need to do trial and error, go for it. Even seasoned developers still get confused about what **this**really points to at times.

If I missed something, or you know of some odd ball scenario that needs a mention, let me know and I can update this article.

Thanks for reading and hope this helped someone out there!

Below part is taken from Bit & Pieces -----------------------------------------------------------------------------

**Arrow Functions vs Traditional functions**

Arrow functions differ from traditional functions in some major ways.

Here’s what the [ECMA specification](https://www.ecma-international.org/ecma-262/6.0/#sec-arrow-function-definitions) says about Arrow functions —

*An****Arrow Function****does not define local bindings for****arguments****,****super****,****this****, or****new.target****. Any reference to arguments, super, this, or new.target within an ArrowFunction****must resolve to a binding in a lexically enclosing environment.***

* **Can’t use arguments special variable.**

The purpose of arguments variable was to expose all the parameters in an array-like object. In ES6, we have [rest parameters](https://codeburst.io/a-simple-guide-to-destructuring-and-es6-spread-operator-e02212af5831) which supercede this task.

* **Can’t be used as a constructor.**

There is no internal method [[Construct]] which allows a normal function to be called by new and no prototype property. Also, Arrow functions do not have their own new.target property, and hence, cannot be used as constructor functions. Rewriting the Person function using arrow functions —

**const Person = name =>** {  
 this.name = name;  
}const person1 = **new** Person('Arfat'); // Will throw an error

Running the above will cause an exception Person is not a constructor.

* **Can’t change the this binding.**

The this value is **always** bound. It cannot be changed. Even by functions which manually reassign the this value, for example, .bind() or .apply().

However, they still have access to Function.prototype and bind(), apply(), and call() still work. The only difference is that they cannot bind a different this parameter value, but they can pass arbitrary arguments, as shown below.

const add = (a, b) => a + b;  
const add5 = **add.bind(null, 5);**  
add5(7); // 12

* **Can’t be used as generator functions**

To use ES6 generators, you have to use function\* syntax.*Learn more about generators*[*here*](https://codeburst.io/understanding-generators-in-es6-javascript-with-examples-6728834016d5)*.*

Despite all these differences, arrow functions and traditional functions both have same typeof value and prototype value.

typeof (**() => {}**) === "function"; // trueObject.getPrototypeOf(**() => {}**) === Function.prototype; // true

Freecodecamp---------------------------------------------------------------------------------------------

“In classic function expressions, the this keyword is bound to different values based on the *context* in which it is called. with arrow functions however, this is*lexically bound*. it means that it uses this from the code that contains the arrow function.”

WESBOS------------------------------------------------------------------------------------------------------------------------------------------

### **When you should not use Arrow Functions**

After learning a little more about arrow functions, I hope you understand that they do not replace regular functions.

Here are some instances where you probably wouldn’t want to use them:

**#1 Object methods - When we want to add function as a property in object literal and use object in it**

When you call cat.jumps, the number of lives does not decrease. It is because this is not bound to anything, and will inherit the value of this from its parent scope which in this case is the window.

var cat = {

lives: 9,

jumps: () => {

this.lives--;

}

}

**#2 Callback functions with dynamic context**

First of all, I’ve got this big button that says ‘Push me’:

<style>

button {font-size: 100px; }

.on {background: #ffc600;}

</style>

<button id="pushy">Push me</button>

When someone pushes or clicks that button, I want to toggle the class of on which should turn it yellow. When someone clicks that button, I’m going to run this following function:

const button = document.querySelector('#pushy');

button.addEventListener('click', () => {

this.classList.toggle('on');

});

But if we click it, we get an error in the console: TypeError, cannot read property 'toggle' of undefined

What does that mean? Well, if we remember from earlier, it’s the browser’s window attribute, right? We can use console.log to confirm it:

const button = document.querySelector('#pushy');

button.addEventListener('click', () => {

console.log(this); *// Window!*

this.classList.toggle('on');

});

Remember: we talked about that if you use an arrow function, the keyword this is not bound to that element. If we use a regular function, the keyword this will be bound to the element we clicked!

const button = document.querySelector('#pushy');

button.addEventListener('click', function() {

console.log(this);

this.classList.toggle('on');

});

In the console, this is now our button, and our big yellow button is actually working. The same rules apply with jQuery, Google Maps or any other DOM Library you are using.

Another example:

var button = document.getElementById('myButton');

button.addEventListener('click', () => {

console.log(this === window); // => true

this.innerHTML = 'Clicked button';

});

With arrow function,

var button = document.getElementById('myButton');

button.addEventListener('click', function() {

console.log(this === button); // => true

this.innerHTML = 'Clicked button';

});

### **#3 Prototype Methods**

As our third example, we’ll talk about when you need to add a prototype method.

class Car {

constructor(make, colour) {

this.make = make;

this.colour = colour;

}

}

Here, I’ve got a class. We haven’t learned about classes yet, but just know that this is a way for us to make new cars.

I have a class constructor where, when you call new Car we pass it the type of Car, as well as the colour of the Car.

I can say beemer is a BMW that is blue, and the subie is a Subaru that is white:

const beemer = new Car('BMW', 'blue');

const subie = new Car('Subaru', 'white');

Let’s go ahead and look at them by calling them in the console, you’ll see that subie comes back as Car {make: "Subaru", colour: "white"}, and beemer will come back as Car {make: "BMW", colour: "blue"}, which is what we’d expect.

Now, after the fact, I added on this prototype method:

Car.prototype.summarize = () => {

return `This car is a ${this.make} in the colour ${this.colour}`;

};

JavaScript

…and what that allows us to do is that, even after these things have been created, we can add methods onto all of them. So our Car.prototype.summarize method is set, so let’s type into the console: subie.summarize.

If you’re using Chrome’s console, you’ll see that it auto-completes the method, because it’s available to you. Even though we added it after we created the Car, because I added it to the prototype, it’s available in every object that has been created from there.

What this prototype does is it returns this.make which is the make that we passed in, and this.color in a sentence.

However, with our example, this.car is undefined and the colour is undefined. Why is that?

It’s because we try to be cool. We try to be a bit of a hot shot here by using an arrow function. Again, why don’t we use an arrow function here? Because we explicitly need the keyword this so you have to use a regular function:

Car.prototype.summarize = function() {

return `This car is a ${this.make} in the colour ${this.colour}`;

};

JavaScript

Now, if we call subie.summarize, it says it’s a white Subaru, and beemer.summarize, we get BMW in blue.

Again, you must use a regular function for that.

#4 **Invoking constructors**

this in a construction invocation is the newly created object. When executing new Fn(), the context of the constructor Fn is a new object: this instanceof Fn === true.

this is setup from the enclosing context, i.e the outer scope which makes it not assigned to newly created object.

var Message = (text) => {

this.text = text;

};

// Throws "TypeError: Message is not a constructor"

var helloMessage = new Message('Hello World!');