SEP 10 2016

Before you start going absolutely bananas on using arrow functions everywhere, we need to chat. **Arrow functions don't replace regular functions**. Just like Flexbox and floats, pixels and rems and anything else new that comes along, the older thing still retains lots of utility because it works differently than the new thing.

We talked about the benefits of ES6 Arrow Functions in earlier videos and blog posts but let's go through a couple examples of when you probably *don't want an arrow function*. All of these are just going to boil down to not having the keyword `this`, but they are also different use cases that you'd run into.

#1 — click handlers

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

Markup

```
<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:

JavaScript

```javascript
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:

JavaScript

```javascript
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!

JavaScript

```javascript
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 sames rules apply with jQuery, Google Maps or any other DOM Library you are using.

#2: Object Methods

Now, let's take a look at this next one, when you need a method to bind to an object.

JavaScript

```javascript
const person = {
    points: 23,
    score: () => {
        this.points++;
    }
}
```

We have our method called `score`, and whenever we call `person.score`, it should add one to our `points`, which is currently 23.

If we run `person.score();` a few times, we should be at 26 or something.

But if I call `person`, `points` is still at 23. Why?

Because it's trying to add points to the window! Remember, when using an arrow function `this` is not bound to anything and it just inherits it from the parent scope which in this case is the window.

So let's do the same thing with an OG function:

JavaScript

```
const person = {
    points: 23,
    score: function()  {
        this.points++;
    }
}
```

There we go. That will actually work, because that's a full on function, and not an arrow function.

3: Prototype Methods

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

JavaScript

```
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`:

JavaScript

```
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:

JavaScript

```javascript
Car.prototype.summarize = () => {
    return `This car is a ${this.make} in the colour ${this.colour}`;
};
```

…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:

JavaScript

```javascript
Car.prototype.summarize = function() {
    return `This car is a ${this.make} in the colour ${this.colour}`;
};
```

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: When you need an arguments Object

For our last example, this is a little bit different:

JavaScript

```javascript
const orderChildren = () => {
    const children = Array.from(arguments);
    return children.map((child, i) => {
        return `${child} was child #${i + 1}`;
    })
    console.log(arguments);
}
```

It doesn’t have to do with the keyword “this,” but we don’t have access to the `arguments` object when you use an arrow function.

This is helpful for when you want to run a function like `orderChildren` here, which can take unlimited arguments.

It might take one, it might take 100. It's going to just say "This child was born #1", or whichever.

For an example, let's type into the console `orderChildren('jill', 'wes', 'jenna')`, which passes in `jill` as our first argument, `wes`, as our second, and `jenna` as our third. When you run it, you'll get an error: `ReferenceError, arguments is not defined`.

this is because `arguments` is a keyword that we have in our `orderChildren` that's going to give us an `Array` or array-ish value of everything that was passed in.

However, you do not get the `arguments` object if you use an arrow function. When you use a regular function, which is going to give us the actual content that we need.

JavaScript

```javascript
const orderChildren = function() {
    const children = Array.from(arguments);
    return children.map((child, i) => {
        return `${child} was child #${i + 1}`;
    })
    console.log(arguments);
}
```

Note: Another fix for this is to use a `...rest` param to collect all the arguments into an array. We will learn all about that in the rest videos and blog posts!

Again, to go through all those really quickly. Make sure that you aren't just using arrow functions willy-nilly. In general, if you do not need the `arguments` object or you do not need `this`, or you know that you will not need it in the future, then you can feel free to go ahead and use an arrow function on everything else.

This entry was posted in ES6, JavaScript. Bookmark the permalink.

9 Responses to *When Not to use an Arrow Function*

Strajk *says:*
September 11, 2016 at 12:56 pm

Alternative to first example is explicitly using `ev` argument

“`

```
button.addEventListener('click', (ev) => {
ev.target.classList.toggle('on');
});
```
“`

Reply

Strajk *says:*
September 11, 2016 at 12:56 pm

Oh, formatting no work

Reply

wesbos *says:*
September 11, 2016 at 1:14 pm

Use currentTarget instead of target as target can change if you have nested elements like a span inside a link

Reply

MaxArt *says:*

September 11, 2016 at 2:14 pm

Object and prototype methods should take the form

```
const person = {
points: 23,
score() {
this.points++;
}
}
```

They're actually "classic" functions and they're also quite nice to read.

Reply

Steven Yap *says:*

November 5, 2016 at 11:55 pm

Is

```
score() {
this.points++;
}
```

the equivalent of

```
score:
function() {
this.points++;
}
```

?

Reply

Michael Connor *says:*

March 1, 2017 at 9:02 am

running person object in chrome inspector, person.points gives expected results. But person.score() gives me undefined.

Reply

Sil *says:*

October 4, 2017 at 3:42 pm

"However, with our example, this.car is undefined and the colour is undefined. Why is that?" <– this.make instead of this.car.

Cheers!

Reply

Anh Tran *says:*

February 8, 2018 at 5:34 am

Awesome post! "this" seems to be the most headache problem with JavaScript!

Reply

gustaf *says:*

January 10, 2019 at 5:32 am

Nice Example…arrow function doesnt have its own this and arguments, so we need to fully understand when we need and when we dont

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Post Comment