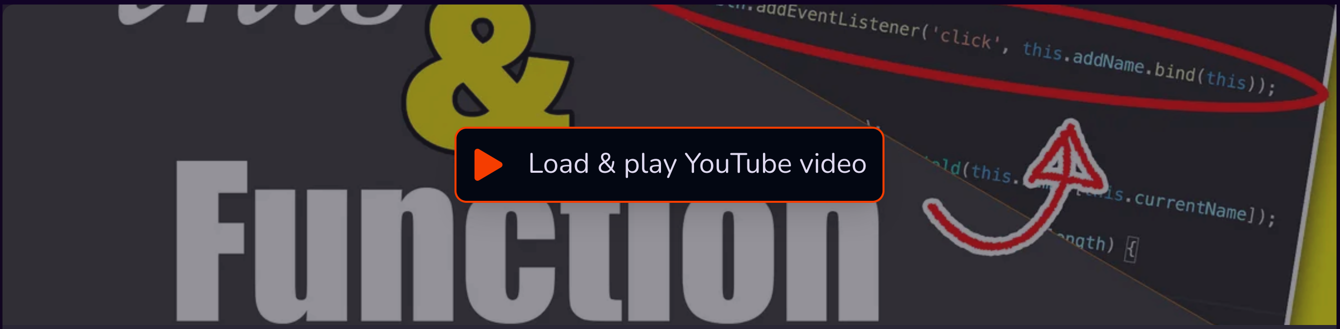
ACADEMIND



'this' Keyword & Function References

The 'this' keyword can lead to a lot of confusion in JS. So can function calls without parenthesis. What's the idea behind this.someFunction.bind(this)?

Created by Maximilian Schwarzmüller

September 11, 2018

Watch Video

# What's the Issue?

Did you ever see a line of JavaScript code that looked something like this?

```
button.addEventListener('click', this.addItem.bind(this));
```

What's this `bind(this)` thing here? And why are the function parentheses missing?

Why does the line not look like this?

```
button.addEventListener('click', this.addItem());
```

Well, this alternative would not work. And here's why.

IMPORTANT

Not a fan of reading articles? Check the video on top of this page!

# Calling Functions & Using Function References

Obviously, you call a function like this in JavaScript:

```
function someFunction() {
  // do something here ...
}
someFunction();
```

And inside an object/ class, you call a method in a similar way:

```
class MyClass {
  constructor() {
    this.myMethod();
  }

  myMethod() {
    // do something here
  }
}
```

Using the above code will execute the methods immediately when the code is run for the first time. In the class, the method `myMethod` is being executed when the constructor is called - i.e. when the class is instantiated.

```
new MyClass(); // this will trigger the constructor and hence call myMetho
```

Sometimes, you don't want to execute a function/ method immediately though.

Consider an event listener on a button:

```
function someFunction() { ... }
const button = document.querySelector('button');

button.addEventListener('click', someFunction());
```

In this snippet, `someFunction` would actually **not** wait for the click to occur but instead also execute right when the code is first parsed/ executed.

That is not what we want though. We just want to "tell JavaScript/ the Browser" that it should execute `someFunction` for us when the button is clicked. This will also ensure that the function can run multiple times - one time for every button click.

What do you do when you want to make sure your friend can visit your parents once he's done with his work for the day?

You don't send him there immediately - instead you tell him where your parents live. This allows your friend to visit them once he got the time. You basically give your friend the address of your parents instead of taking him with you.

The same concept can be used in JavaScript. You can "give JavaScript/ the Browser" the address of something (=> a function) instead of executing it manually right away.

This is done by passing a so called **"reference"**.

For the event listener, the following code passes a reference to the "to-be-executed" function to the event listener (i.e. to the button in this case).

```
function someFunction() { ... };

const button = ...;

button.addEventListener('click', someFunction);
```

Please note, that the parentheses **are missing** after `someFunction`. Therefore, we don't call the function - instead we just pass a pointer to the function (a so called reference) to the event listener (and hence to the button object).

In the context of a JavaScript class, the code looks pretty much the same:

```
class MyClass {
  constructor() {
    const button = ...;
    button.addEventListener('click', this.myMethod);
  }

  myMethod() { ... }
}
```

The `this` keyword is important here though. It basically points at the object that is created based on the class. And since `myMethod` is a method of the class/ object, it can only be accessed via `this`.

That's how you pass a reference to a function instead of calling it immediately. And that's why you would use this syntax without the parentheses.

# When "this" Behaves Strangely

`this` is required to access class methods or properties from anywhere inside of that class/ object.

IMPORTANT

In case classes are brand-new to you, consider taking my ES6 course as I dive deeper into classes there.

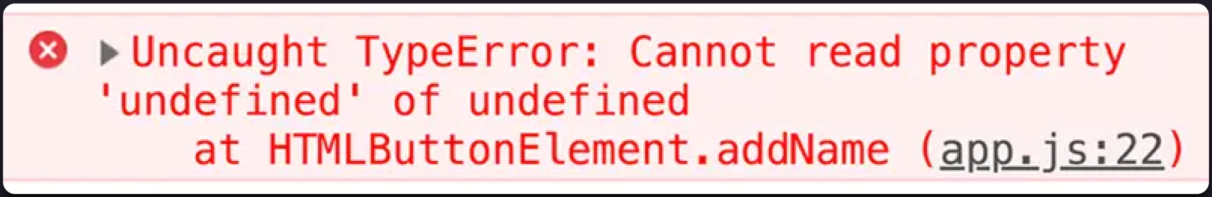
Let's move to a real example - one where `this` will actually lead to a strange behavior.

```
class NameGenerator {
  constructor() {
    const btn = document.querySelector('button');
    this.names = ['Max', 'Manu', 'Anna'];
    this.currentName = 0;
    btn.addEventListener('click', this.addName);
  }

  addName() {
    const name = new NameField(this.names[this.currentName]);
    this.currentName++;
    if (this.currentName >= this.names.length) {
      this.currentName = 0;
    }
  }
}
```

Let's not worry about what `new NameField(...)` does - you can watch the video (at the top of this page) to see the full example. It basically just renders a new `<li>` with the name as text into the DOM.

But let's worry about whether that succeeds or not. Because we'll actually get an error:



```
▶Uncaught TypeError: Cannot read property
'undefined' of undefined
    at HTMLButtonElement.addName (app.js:22)
```

This line is causing the error:

```
const name = new NameField(this.names[this.currentName]);
```

Somehow, accessing `this.names` and `this.currentName` fails here.

But why? Doesn't `this` refer to the object/ class?

Well, it actually doesn't. `this` is not defined to refer to the object that encloses it when you write your code.

Instead, `this` refers to "whoever called the code in which it's being used".

And in this case, the `button` is responsible for executing `addName`.

We can see that, if we log the value of `this` inside of `addName`:

```
addName() {
  console.log(this);
  ...
}
```

This will print:

```
<button>Add Name</button>                                    app.js:21
```

So `this` is now referring to the `<button>` element to which we attached the `click` event listener.

That is actually the default JavaScript behavior.

`this` refers to whoever called a method that uses `this`.

Obviously, this is not the behavior we want here - and thankfully, you can change it.

You can bind `this` inside of `addName` to something else than the button. You can bind it to the surrounding class/ object:

```
btn.addEventListener('click', this.addName.bind(this));
```

`bind()` is a default JavaScript method which you can call on functions/ methods. It allows you to bind `this` **inside** of the "to-be-executed function/ method" to any value of your choice.

In the above snippet, we bind `this` inside of `addName` to the same value `this` refers to in the constructor.

In that constructor, `this` will refer to the class/ object because we execute that code on our own. The constructor essentially is always executed by the object itself you could say, hence `this` inside of the constructor also refers to that object.

`bind` would also allow you to pass arguments to the function you'll eventually call but you can learn more about it here.

Is the article helpful?

# Summary

That's it!

This hopefully illustrates why you can have code where you "call functions" (not really) without adding parentheses and why you may have to use `bind(this)` to make `this` work correctly in that function/ method.

Definitely check out my two JavaScript courses if you want to learn way more about JavaScript:

- JavaScript - The Complete Guide
- My Accelerated ES6 Training

Or dive into one of our many other courses - huge discounts await you!

Enter your email address

By registering, you confirm that you read & accepted our privacy policy.

Register

COURSES

All Courses

Academind Pro

OTHER RESOURCES

Tutorial Articles

Podcast

Academind Community

SUBSCRIBE TO OUR NEWSLETTER

Be the first to be informed about our new courses, articles & more.

Enter your email address

By registering, you confirm that you read & accepted our privacy policy.

Register