How to fix "this is undefined" in Vue

You're happily coding along, loving how awesome Vue is, when it strikes.

Your VueJS app doesn't work, and you get an error that says:

```
this is undefined
```

Don't worry, you're not alone — I've run into this issue countless times, and I'll show you just how to solve it.

The likely cause of this is that you're mixing up your usage of regular functions and arrow functions. My guess is that you're using an arrow function. If you replace the arrow function with a regular function it will probably fix the issue for you.

But let's go a little further and try to understand **why this works**.

After all, knowledge is power, and if you know what caused your problem, you'll be able to avoid a lot of frustration and wasted time in the future.

There are also a few other places where you can get tripped up with this error:

- Fetching data using `fetch` or `axios`
- Using libraries like `lodash` or `underscore`

I'll cover these as well, and how to do them properly.

Understanding the two main types of functions

In Javascript we get two different kinds of functions. They operate in almost identical ways, except they differ in how they treat the variable `this`.

This causes a ton of confusion for new and old Javascript devs alike — but by the time we're through you won't be caught by this one anymore.

Regular functions

A regular function can be defined in a few different ways.

The first way is less common in Vue components, because it's a bit longer to write out:

```
methods: {
  regularFunction: function() {
    // Do some stuff
  }
}
```

The second way is the shorthand function, which is probably more familiar to you:

```
methods: {
  shorthandFunction() {
    // Do some stuff
  }
}
```

In a regular function like this one, `this` will refer to the "owner" of the function. Since we're defining it on the Vue component, `this` **refers to your Vue component**. I'll explain how this scoping works in more detail later on.

In most cases you should use a regular function with Vue, especially when creating:

- methods
- computed props
- watched props

While regular functions are usually what you need, arrow functions come in very handy as well.

Arrow functions

Arrow functions can be even shorter and quicker to write, and have gained lots of popularity recently because of this. However, they aren't too different when defining a method on an object like we are doing when writing Vue components.

This is what they look like on a Vue component:

```
methods: {
  arrowFunction: () => {
    // Do some stuff
  }
}
```

The real differences start to come in to play when dealing with how `this` works.

In an arrow function, `this` **does not** refer to the owner of the function.

An arrow function uses what is called **lexical scoping**. We'll get into this more in a bit, but it basically means that the arrow function takes `this` from it's context.

If you try to access `this` from inside of an arrow function that's on a Vue component, you'll get an error because `this` doesn't exist!

```
data() {
  return {
    text: 'This is a message',
  };
},
methods: {
  arrowFunction: () => {
    console.log(this.text);  // ERROR! this is undefined
  }
}
```

So in short, try to avoid using arrow functions on Vue components. It will save you a lot of headaches and confusion.

There **are** times when it's nice to use a short arrow function. But this only works if you aren't referencing `this`:

```
computed: {
  location: () => window.location,
}
```

Now that we know the two main types of functions, how do we use them in the correct way?

Anonymous Functions

Anonymous functions are great for when you just need to create a quick function and don't need to call it from anywhere else. They're called **anonymous** because they aren't given a name, and aren't tied to a variable.

Here are some scenarios where you'd use an anonymous function:

- Fetching data using `fetch` or `axios`
- Functional methods like `filter`, `map`, and `reduce`
- Anywhere else **inside of Vue methods**

I'll show you what I mean:

```
// Fetching data
fetch('/getSomeData').then((data) => {
  this.data = data;
});

// Functional methods
const array = [1, 2, 3, 4, 5];
const filtered = array.filter(number => number > 3);
const mapped = array.map(number => number * 2);
const reduced = array.reduce((prev, next) => prev + next);
```

As you can see from the examples, most of the time when people create anonymous functions they use arrow functions. We typically use arrow

functions for several reasons:

- **Shorter and more condensed syntax**
- Improved readability
- `this` is taken from the **surrounding context**

Arrow functions also work great as anonymous functions inside of Vue methods.

But wait, didn't we just figure out that **arrow functions don't work** when we try to access `this`?

Ah, but here is where the distinction is.

When we use arrow functions **inside a regular or shorthand function**, the regular function sets `this` to be our Vue component, and the arrow function uses that `this` (say that 5 times fast!).

Here's an example:

```
data() {
  return {
    match: 'This is a message',
  };
},
computed: {
  filteredMessages(messages) {
    console.log(this); // Our Vue component

    const filteredMessages = messages.filter(
      // References our Vue Component
      (message) => message.includes(this.match)
    );

    return filteredMessages;
  }
}
```

Our filter can access `this.match` because the arrow function uses the same context that the method `filteredMessages` uses. This method, **because it is a regular function** (and not an arrow function), sets it's own context to be the Vue instance.

Let's expand further on how you would this to fetch data using `axios` or `fetch`.

Using the right function when fetching data

If you're fetching async data using `fetch` or `axios`, you're also using promises. Promises **love** anonymous arrow functions, and they also make working with `this` a lot easier.

If you're fetching some data and want to set it on your component, this is how you'd do that properly:

```
export default {
  data() {
    return {
      dataFromServer: undefined,
    };
  },
  methods: {
    fetchData() {
      fetch('/dataEndpoint')
        .then(data => {
          this.dataFromServer = data;
        })
        .catch(err => console.error(err));
    }
  }
};
```

Notice how we're using a regular function as the method on the Vue component, and then using anonymous arrow functions inside of the promise:

```
.then(data => {
  this.dataFromServer = data;
})
```

Inside of the scope for `fetchData()`, we have that `this` is set to our Vue component because it is a regular function. Since arrow functions use the outer scope as their own scope, the arrow functions also set `this` to be our Vue component.

This allows us to access our Vue component through `this` and update `dataFromServer`.

But what if you need to pass functions to a helper library, like `lodash` or `underscore`?

Using with Lodash or Underscore

Let's say that you have a method on your Vue component that **you want to debounce using Lodash or Underscore**. How do you prevent those pesky `this is undefined` errors here?

If you come from the React world, you've probably seen something similar to this.

Here is how we would do it in Vue:

```
created() {
  this.methodToDebounce = _.debounce(this.methodToDebounce, 500);
},
methods: {
  methodToDebounce() {
    // Do some things here
  }
}
```

That's it!

All we're doing is taking our function, wrapping it in the debounce function, and returning a new one that has the debouncing built in. Now when we call `this.methodToDebounce()` on our Vue component we will be calling the debounced version!

What is lexical scoping?

So I promised that I would explain this more clearly, so here it is.

As I mentioned before, the main reason that there is a difference between regular functions and arrow functions has to do with **lexical scoping**.

Let's break down what it means. We'll work in reverse order.

First, **a scope is any area of the program where a variable exists**. In Javascript, the `window` variable has global scope — it's available everywhere. Most variables though are limited to the function they are defined in, the class they are a part of, or limited to a module.

Second, **the word "lexical" just means that the scope is determined by how you write the code**. Some programming languages will determine what is in scope only once the program is running. This can get really confusing, so most languages just stick with lexical scoping.

Arrow functions use lexical scoping, but regular and shorthand functions do not.

The trickiest part here is how lexical scoping affects `this` in your functions. For arrow functions, `this` is bound to the same `this` of the outer scope. Regular functions have some weirdness to how they bind `this`, which is why arrow functions were introduced, and why most people try and use arrow functions as much as possible.

Examples of how scope works in functions

Here are some examples to illustrate how scope works differently between the two function types:

```
// This variable is in the window's scope
window.value = 'Bound to the window';

const object = {
  // This variable is in the object's scope
  value: 'Bound to the object',
  arrowFunction: () => {
    // The arrow function uses the window's scope for `this`
    console.log(this.value); // 'Bound to the window'
  },
  regularFunction() {
    // The regular function uses the object's scope for `this`
    console.log(this.value);  // 'Bound to the object'
  }
};
```

Now you know a bit about how scope is bound to functions in Javascript!

But there is a way to override this default behaviour and do it all yourself...

Binding scope to a function

Did you know that you can actually **override how `this` is bound** and provide your own `this` to a function?

You need to use the bind method on the function:

```
const boundFunction = unboundFunction.bind(this);
```

This gives you much greater flexibility in writing Vue components, and let's you reuse your methods more easily.

It's a bit more advanced in it's usage, so you should try to avoid using it too often.