Tip 36: Prevent Context Confusion with Arrow Functions

In this tip, you'll learn how to use the arrow function to avoid context errors.

We'll cover the following

Scope, context & `this` keyword

Scope and context are probably the two most confusing concepts for JavaScript developers. A function's scope, at it simplest, is what variables the functions can access. We explored this previously in Tip 3, Isolate Information with Block Scoped Variables. Now you're going to learn about context. **Context** is what the keyword `this` refers to in a function or class.

Not only are both concepts hard to grasp, but people often confuse them. I know I confuse them all the time. Ryan Morr gives a simple way to remember the difference: *Scope* pertains to **functions** and *context* pertains to **objects**. While that's not 100 percent true—you can use `this` in any function—it's a good general rule.

Understanding context

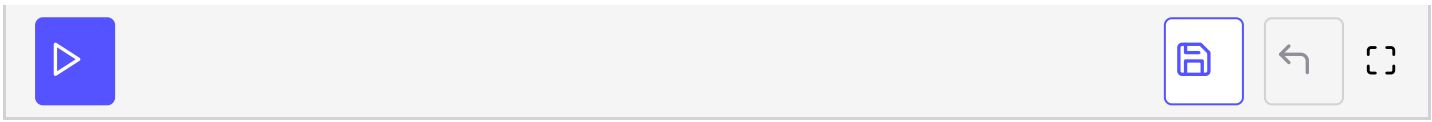
To understand context, start with a very simple object. For example, think about an object called `Validator`, which sets an invalid message on form fields. You have one property, `message`, and one method, `setInvalidMessage()`.

In the `setInvalidMessage()` method, you can refer to the `message` property using `this.message`. To see it in action, call the method from the object.

```
const validator = {
```

```
const validator = {
    message: 'is invalid.',
    setInvalidMessage(field) {

        return `${field} ${this.message}`;
    },
};
console.log(validator.setInvalidMessage('city'));
```



As you see, `this.message` refers to the property on the object. This works because, when the method is called from the object, the function creates a `this` binding with the containing object as context.

Working with `this`

Now before you go any further, you should know that concepts surrounding the keyword `this` are pretty complex. There's a whole book in the You Don't Know JS series on the subject. This book is mandatory reading for JavaScript developers and there's no way to cover the same level of information here in one tip. Instead, you're going to see one of the most common context mistakes.

Working with `this` on objects usually isn't a problem until you try to use a function as *callback* for another function.

For example, you'll encounter problems with `this` when using `setTimeout()`, `setInterval()`, or your favorite array methods such as `map()` or `filter()`. Each of these functions takes a callback, which, as you'll see, *changes* the context of the callback.

What do you think will happen if you try to refactor your `setInvalidMessage()` method to take an array of fields using `map()` to add the message? The code change isn't complicated. Create a new method called `setInvalidMessages()` that maps over an array adding the message to each.

```
const validator = {
    message: 'is invalid.',
    setInvalidMessages(...fields) {
        return fields.map(function (field) {
            return `${field} ${this.message}`;
        });
    },
};
```

The problem is that when you invoke the function, you'll get either a `TypeError` or

`undefined`. This is where most developers get frustrated and refactor the code to remove a reference to `this`.

```
const validator = {
    message: 'is invalid.',
    setInvalidMessages(...fields) {
        return fields.map(function (field) {
            return `${field} ${this.message}`;
        });
    },
};
console.log(validator.setInvalidMessages('city'));
```

Think for a moment about what may cause this problem. Remember that whenever you call a function, it creates a this binding based on where it's called. `setInvalidMessage()` was called in the context of an object. The `this` context was the object. The callback for the `map` function is called in the *context* of the `map()` method, so the `this` binding is no longer the `Validator` object. It will actually be the *global* object: `window` in a browser and the Node.js environment in a REPL. The callback doesn't have access to the message property.

Arrow functions & `this`

This is where arrow functions come in. Arrow functions *don't* create a new `this` binding when you use them. If you were to rewrite the preceding `map()` callback using an arrow function, everything would work as expected.

```
const validator = {
    message: 'is invalid.',
    setInvalidMessages(...fields) {
        return fields.map(field => {
            return `${field} ${this.message}`;
        });
    },
};
console.log(validator.setInvalidMessages('city'));
```

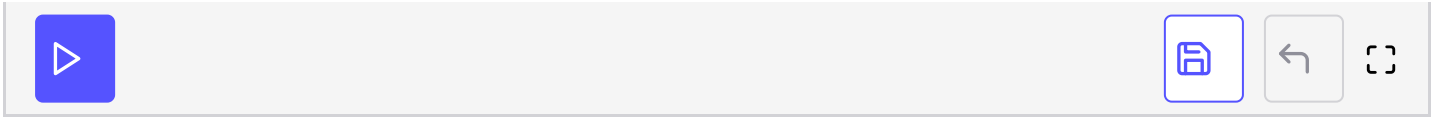
Now this may seem great and a good reason to always use arrow functions. But remember, sometimes you actually do want to set a `this` context.

For example, what if you wrote your original `setInvalidMessage()` method not as a named method but as an arrow function assigned to a property?

```
const validator = {
    message: 'is invalid.',
    setInvalidMessage: field => `${field} ${this.message}`,
};
```

You'd have the exact same `TypeError` or `undefined` when you called it.

```
const validator = {
    message: 'is invalid.',
    setInvalidMessage: field => `${field} ${this.message}`,
};
console.log(validator.setInvalidMessage('city'));
```



In this case, you didn't create a new `this` context binding to the current object. Because you didn't create a new context, you're still bound to the *global* object.

To summarize, arrow functions are great when you already have a context and want to use the function inside another function. They're a problem when you need to set a new `this` binding.

This isn't the last you'll see of `this`. It plays a big part in classes, and context bindings will come up again in Tip 42, Resolve Context Problems with Bind().

The next chapter explores classes in JavaScript. If you come from an object-oriented background, you'll see a lot that looks familiar and a lot that you won't expect.