Tip 42: Resolve Context Problems with Bind()

In this tip, you'll learn how to solve this errors with bind().

We'll cover the following

- Preventing context problems
  - Solution 1: Using arrow functions
  - Solution 2: Using bind()

Preventing context problems

In Tip 36, Prevent Context Confusion with Arrow Functions, you saw how functions create a new context and how a new context can give you results you aren't expecting. Changing context can create confusion, particularly when you're using the `this` keyword in callbacks or array methods.

Sadly, the problem doesn't go away in classes. Earlier, you learned how you can use arrow functions to create another function without a new context. In this tip, you'll learn more techniques for preventing context problems. The techniques you're about to learn work on object literals and classes, but they're much more common in class syntax, particularly if you're using libraries such at React or Angular.

Think back to the original example, a validator. Originally, you made it as an object literal, but now that you know a bit about classes, you can make it a class. The class will have one property, a message, and two methods: `setInvalidMessage()`, which returns a single invalid message for a field, and `setInvalidMessages()`, which maps an array of fields to a series of invalid messages.

```
class Validator {
    constructor() {
        this.message = 'is invalid.';
    }
    setInvalidMessage(field) {
        return `${field} ${this.message}`;
    }
    setInvalidMessages(...fields) {
        return fields.map(this.setInvalidMessage);
```

```
    }
}
```

All you did was translate an object with properties and methods to a class with properties and methods.

The `Validator` class will have the exact same context problem as your object. When you call `setInvalidMessages()`, the function creates a `this` binding to the class. Inside the method, you call `map()` on an array and pass `setInvalidMessage()` as the callback. When the `map()` method invokes `setInvalidMessage()`, it will create a new `this` binding in the context of the array method, not the class.

```
class Validator {
    constructor() {
        this.message = 'is invalid.';
    }
    setInvalidMessage(field) {
        return `${field} ${this.message}`;
    }
    setInvalidMessages(...fields) {
        return fields.map(this.setInvalidMessage);
    }
}
const validator = new Validator();
validator.setInvalidMessages('city');
// TypeError: Cannot read property 'message' of undefined
```

Context problems are common in the React community. Nearly every class has some form of binding problem. Cory House has a great breakdown of different ways to solve the binding problem in React. You'll be seeing an adaptation of most of those solutions in a more generic class.

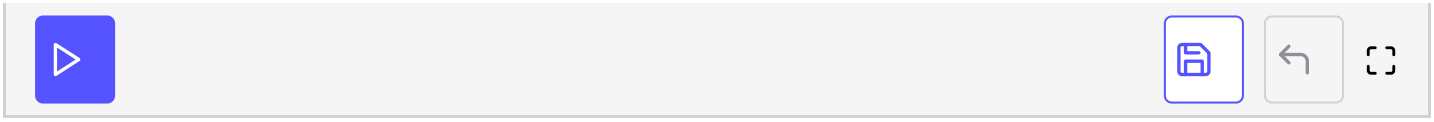
Solution 1: Using arrow functions

The first way to solve the problem is the same as the solution suggested in Tip 36, Prevent Context Confusion with Arrow Functions. Convert your method to an arrow function. The arrow function won't create a new this binding and it won't throw an error.

The only downside to this approach is that when you're working with class syntax, you'll have to move your function to a property rather than a method. It's not a big deal on objects because objects and properties both use a keyvalue declaration. In classes, you have to set properties in the constructor and the method will look a

little out of place. Now you're stuck with a situation where some methods are set in the constructor and some are set as class methods.

```
class Validator {
    constructor() {
        this.message = 'is invalid.';
        this.setInvalidMessage = field => `${field} ${this.message}`;
    }
    setInvalidMessages(...fields) {
        return fields.map(this.setInvalidMessage);
    }
}
const v = new Validator();
console.log(v.setInvalidMessages('city'));
```



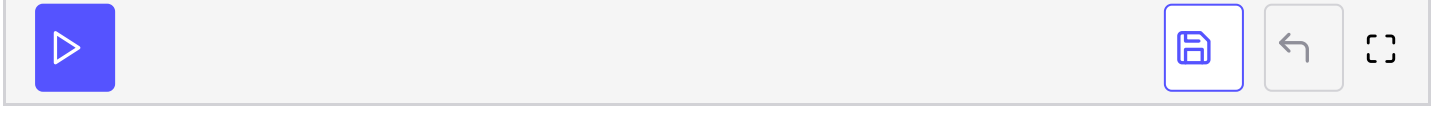
Moving the method to a property in the constructor may solve your context problem, but it creates another. Methods are defined in multiple places. And depending on how many methods you create this way, your constructor can get large quickly.

Solution 2: Using `bind()`

A better solution is to use the `bind()` method. This method exists on all functions and lets you state your context explicitly. You'll always know what `this` refers to because you tell the function exactly where to bind.

As an example, suppose you have a function that refers to a property on `this`. The function doesn't actually have that property. The property `this` refers to may not yet exist. There's no rule that says properties must exist when you declare a function. But they do need to exist at *runtime* when you call a function or else you'll get `undefined`. With this function, you can explicitly set this to a specific object using `bind`.

```
function sayMessage() {
    return this.message;
}
const alert = {
    message: 'Danger!',
};
const sayAlert = sayMessage.bind(alert);
console.log(sayAlert());
```



Whenever the function uses this, it will lock in the object you bound to it. Kyle Simpson calls this explicit binding because you're declaring the context and not relying on the engine to set it at runtime.

In the preceding example, you're binding the `sayMessage()` function explicitly to an object that has the message property.

Now it's time for things to get a little confusing. You can also bind a function to the *current context* by binding it to `this`. It may seem odd to bind a `this` to, well, this, but all you're doing is telling the function to use the current context rather than creating a new one. Unlike an arrow function, the function is still creating a `this` binding—it's just using the current binding rather than building a new one.

In your `Validator` class, you can bind the function to the current context before you pass it to the `map()` method.

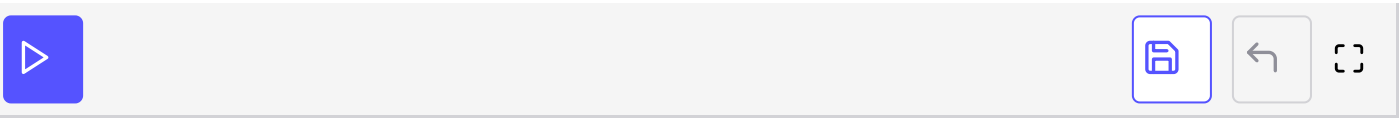
```
class Validator {
    constructor() {
        this.message = 'is invalid.';
    }
    setInvalidMessage(field) {
        return `${field} ${this.message}`;
    }
    setInvalidMessages(...fields) {
        return fields.map(this.setInvalidMessage.bind(this));
    }
}
const v = new Validator();
console.log(v.setInvalidMessages('city'));
```

That's a fine approach. The only downside is if you use the function in another method, you'll have to bind it again. A lot of developers avoid multiple binds by setting a bound method to a property of the same name in the constructor.

This is very similar to creating an arrow function in the constructor. The advantage is that your methods are still defined in the same place. They're merely bound in the constructor. Now you define all your methods in one place, the body. You declare your properties in another place, the constructor. And you set your context in one place, also the constructor.

```
class Validator {
    constructor() {
        this.message = 'is invalid.';
```

```
        this.setInvalidMessage = this.setInvalidMessage.bind(this);
    }
    setInvalidMessage(field) {
        return `${field} ${this.message}`;
    }
    setInvalidMessages(...fields) {
        return fields.map(this.setInvalidMessage);
    }
}
const v = new Validator();
console.log(v.setInvalidMessages('city'));
```



Both approaches—*using arrow functions and binding a function to* `this` — work with the current spec. In a future spec, you'll be able to set class properties outside of the constructor. With the new spec, you assign arrow functions to properties alongside other method definitions. It's the best of both worlds.

```
class Validator {
    message = 'is invalid.';
    setMessage = field => `${field} ${this.message}`;
    setInvalidMessages(...fields) {
        return fields.map(this.setMessage);
    }
}
const v = new Validator();
console.log(v.message); //'is invalid'
```

As with other proposed specs, you can use this feature right now with the proper Babel plugin. This particular feature isn't currently supported in any version of Node.js, so you won't be able to try it out in the REPL.

As with other context problems, try not to get too hung up on the details. Binding will make more sense when you see it organically. Just remember: If you're encountering unexpected behaviors or weird errors when using this, you might want to explicitly bind the context. Until that point, don't worry. Binding can be expensive, and you really should only use it when you need to solve a specific problem.

At this point, you should be able to create and extend classes with ease. Despite the controversy, it makes writing JavaScript a lot more intuitive for those outside the language and more succinct for those who've been developing JavaScript for years.

In the next chapter, you'll learn how to work with data outside your code by

exploring promises, fetch methods, and asynchronous functions.