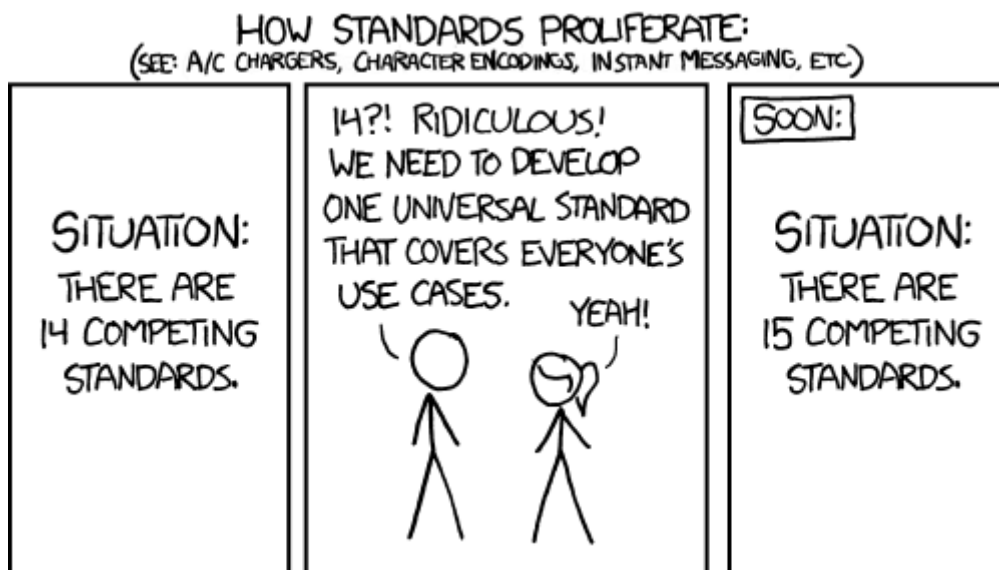


# Arrow functions are disrupting React.Components



Karthik Kalyanaraman

Jan 14 · 6 min read



Ever wondered why we 'bind' class methods to 'this' inside constructors in React Components? And, after a while, when your component becomes big with a lot of call back functions, the constructor looks ugly and cumbersome like this,

```
class Foo extends Component {
  constructor(props) {
    this.cb1 = this.cb1.bind(this)
    this.cb2 = this.cb2.bind(this)
    this.cb3 = this.cb3.bind(this)
  }
  cb1(){}
  cb2(){}
  cb3(){}
  render() {
    <Button onClick={this.cb1}>
      click me
    </Button>
  }
}
```

Recently, one of my friends from the open-source community suggested using arrow functions instead. Reason? With arrow functions, you don't have to bind 'this' inside the constructor like this,

```
class Foo extends Component {
  constructor(props) {
  }
  cb1 = () => {}
  cb2 = () => {}
  cb3 = () => {}
  render() {
    <Button onClick={this.cb1}>
      click me
    </Button>
  }
}
```

This is great! This works like a charm and looks much cleaner than the previous example. But, I decided to unravel this mystery to really understand where this is coming from.

Before understanding why and how this works, let's roll back a little bit to the fundamentals. So what is 'bind'?

---

**“The `bind()` method creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.”**

---

Let's look at an example,

```
function foo() {
  console.log(this.name)
}
var man = {"name": "jack"}

foo() // undefined

// bind man to foo
foo = foo.bind(man)

foo() // jack
```

So, before binding man to foo, foo being a regular function, has no reference to 'name' and hence, it doesn't print anything when you call this.name. But, when you bind man to foo object, foo now has a 'this' reference(which is 'man') and it prints the 'name' from the man object.

Now, that we have sorted 'bind' out, let's move one step further and understand what happens in a special situation,

```
function foo() {
  console.log(this.name)
}
var man = {"name": "jack"}
foo() // undefined

// Attach foo to man as a property
man.foo = foo
man.foo() // jack

// Copy man.foo to a new var bar
var bar = man.foo
bar() // undefined (WTF?)
```

When we attach foo to man and call it on man, obviously now foo sees man's name and prints 'jack' correctly. But what happened when we create a new variable bar and copy man.foo to bar? All of a sudden, man.foo loses 'this'. Because? 'bar' sees a different 'this' which is the 'window' object of the document and hence, there is no 'name' defined here. How do we overcome this issue? That's correct! We use 'bind'

```
foo = foo.bind(man)
man.foo = foo

var bar = man.foo
bar() // jack
```

Okay! This works. But, what happens in classes? Remember, JavaScript classes are syntactic sugar for JavaScript constructor functions. Which means, the behavior is exactly the same in classes.

But, why do I even copy a class method to a new variable? Doesn't make sense right? But, this is exactly what happens when you create...

## “Callbacks”

Remember that onClick property of the button component? That’s right!

```
class Foo extends React.Component {
  constructor() {
    this.clickhandler = this.clickhandler.bind(this)
  }
  clickhandler() {
    console.log("you clicked me!!")
  }
  render() {
    return(
      <div>
        <button onClick={this.clickhandler}> // => CALLBACK
          Click me
        </button>
      </div>)
    )
  }
}
```

Phew! Took a while to get that connection right? Now, it all makes sense. Okay! So, what about arrow functions? Why aren’t we binding them and how do they just work fine?

Back to basics!

```
var name = "rose"
var foo = () => { console.log(this.name) }
foo() // rose

var man = {"name": "jack"}
man.foo = foo
man.foo() // rose

foo.bind(man)
man.foo() // rose
```

Why am I not able to bind or attach man’s ‘name’ property to foo at all?

**“This is because, you cannot “rebind” an arrow function. It will always be called with the context in which it was defined.”**

This also means that, arrow functions as callbacks will work just fine.

Because, the callback arrow function, clickhandler is permanently defined in the context, which is the context of class Foo. But, regular functions lose the context, when you assign it to a variable or in other words, use it as a callback.

```
class Foo extends React.Component {
  constructor() {
  }
  clickhandler = () => {
    console.log("you clicked me!!")
  }
  render() {
    return(
      <div>
        <button onClick={this.clickhandler}> // => CALLBACK
          Click me
        </button>
      </div>)
    )
  }
}
```

But, what do we lose? We always trade something off to gain something right? Let's peek into both the definitions.

```
// Assume FooRegularFunction defines clickhandler as regular
function and FooArrowFunction defines clickhandler

console.dir(FooRegularFunction)
console.dir(FooArrowFunction)
```

We see that, clickhandler gets assigned to prototype property of the FooRegularFunction(Assume App class and FooRegularFunction class are same on the image).

```
▼ f App(props) ⓘ
  arguments: (...)
  caller: (...)
  length: 1
  name: "App"
  ▼ prototype: Component
    ► clickhandler: f clickhandler()
    ► constructor: f App(props)
```

```
    isMounted: (...)  
    ▶ render: f render()  
    replaceState: (...)  
    ▶ __proto__: Object  
    ▶ __proto__: f Component(props, context, updater)  
    [[FunctionLocation]]: App.js:6  
    ▶ [[Scopes]]: Scopes[3]
```

---

>

FooRegularFunction

But, for FooArrowFunction, the clickhandler is nowhere to be seen. It is not assigned to the prototype property. But, where is it anyway?

```
▼ f App(props) ⓘ  
  arguments: (...)  
  caller: (...)  
  length: 1  
  name: "App"  
  ▼ prototype: Component  
    ▶ constructor: f App(props)  
      isMounted: (...)  
    ▶ render: f render()  
      replaceState: (...)  
    ▶ __proto__: Object  
    ▶ __proto__: f Component(props, context, updater)  
      [[FunctionLocation]]: App.js:6  
    ▶ [[Scopes]]: Scopes[3]
```

---

>

FooArrowFunction

To understand this, we are going to understand another subtle topic.

```

class Foo {
  constructor(name) {
    this.name = name
  }
  function bar() {console.log(this.name)}
  foobar = () => {console.log(this.name)}
}

var f = new Foo("jack")
f.bar() // jack
f.foobar() // jack

// <see images below>
console.dir(Foo)
console.dir(f)

```

In the above example, we have two class methods, one, defined as a regular function and the other defined as an arrow function. Let's look at the class and the instance's properties.

```

▼ f Foo(name) ⓘ
  arguments: (...)
  caller: (...)
  length: 1
  name: "Foo"
  ▼ prototype:
    ► bar: f bar()
    ► constructor: f Foo(name)
    ► __proto__: Object
    ► __proto__: f ()
      [[FunctionLocation]]: App.js:41
    ► [[Scopes]]: Scopes[3]

```

---

```

▼ Foo ⓘ
  ► foobar: f ()
    name: "jack"
  ▼ __proto__:
    ► bar: f bar()
    ► constructor: f Foo(name)

```

## ► `__proto__`: Object

---

In the above image, first part is Foo class properties and second part is Foo instance properties. As expected, `bar()` which is a regular function gets attached to the prototype of the class definition and gets copied on to `__proto__` property when you create an instance. But, `foobar()` is hidden in Foo class properties, but gets defined as an independent property on the instance.

This is precisely the reason why arrow functions as class methods work just fine. But, also get hidden in the class properties. But, why is it hidden though?

Enter ES Next. You must understand that, this is an experimental feature (stage 3) proposed at TC39, the JavaScript standards committee. And this is proposed precisely to bring the great benefits of the super-star feature of Object Oriented Programming, which is , *“Encapsulation”*.

But, I will save that topic for another day.

I would like to end this story by stating,

**“Replacing class methods defined as regular functions with arrow functions effectively makes the class methods private(not entirely true). The pros and cons of this approach is purely subjective on a case by case basis.”**

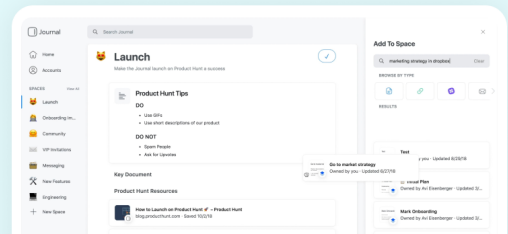
Hope you enjoyed reading this story. If you found this helpful, I would definitely appreciate an applause via the applause button. It helps others to discover this story.






**Read next:** What would be possible if all our thoughts were connected and easily accessible?

Meet Journal →



 Read this story later in Journal.

 Wake up every Sunday morning to the week's most noteworthy Tech stories, opinions, and news waiting in your inbox: Get the noteworthy newsletter >

[JavaScript](#)   [Js](#)   [React](#)   [React Native](#)   [Reactjs](#)

[About](#)   [Help](#)   [Legal](#)