WTF IS THIS?

A BRIEF HISTORY OF JAVASCRIPT

April 1995 - Brendan Eich hired by Netscape to design a scripting language for the browser

10 days later - JavaScript is born (called Mocha at the time)

“If I had done classes in JavaScript back in May 1995, I would have been told that it was too much like Java or that JavaScript was competing with Java … I was under marketing orders to make it look like Java but not make it too big for its britches … [it] needed to be a silly little brother language.”

CONTEXT

The value of `this` is equal to `boundValue || context`.
Plain functions do not have a bound value initially.

The context is **not** defined or known at declaration; it is provided when *calling* the function.

```
function foo() { // this = ?
  return this.bar;
}
```

CALLING A BARE FUNCTION

```
function foo() { // this = ?
  return this.bar;
}
```

When the function is called on its own, the context is the global object. In the browser, the global object is `window`. In Node, it is `global`.

```
var bar = 1;
foo(); // context = window; returns 1
```

CALLING FROM AN OBJECT

```
function foo() { // this = ?
  return this.bar;
}

const obj = { foo, bar: 4 };
obj.foo(); // context = obj; returns 4
```

FORCING CONTEXT

Functions have a `.call` method which can be used to provide a context without the function being a property on that object

```
function foo() { // this = ?
  return this.bar;
}

const obj = { bar: 4 };
foo.call(obj); // context = obj; returns 4
```

PITFALLS

```
const obj = {
  bar: 6,
  foo() { // this = ?
    return this.bar;
  },
};

const foo = obj.foo;
foo(); // context = window; returns undefined

obj.foo(); // context = obj; returns 6
```

BINDING

Binding a function creates a *new* function which bakes in a provided value to be used as `this`.

A bound function cannot be unbound or have the bound value overridden in any way.

The value of `this` is equal to `boundValue || context`. The context from which a bound function is called is ignored in favor of the baked in value.

BINDING A PLAIN FUNCTION

Functions have a `.bind` method which can be used to create a bound function

```
function foo() { // this = ?
  return this.bar;
}

const obj = { bar: 4 };
const foo2 = foo.bind(obj); // boundValue = obj
```

```
foo(); // context = window; returns undefined

foo2(); // boundValue = obj; returns 4
```

```
obj.bar = 5;
foo2(); // boundValue = obj; returns 5
```

ARROW FUNCTIONS

Arrow functions are "auto-binding". Unlike plain functions, the definition of an arrow function is significant.

When an arrow function is declared, it is automatically bound with the context of its declaration.

AUTO-BINDING

```
const foo = () => this.bar; // boundValue = window
var bar = 4;

foo(); // boundValue = window; returns 4

foo.call({ bar: 8 }) // boundValue = window; returns 4
```

```
const obj = {
  foo: () => this.bar, // boundValue = obj
  bar: 5,
};

const foo = obj.foo;

foo(); // boundValue = obj; returns 5
```

CLASSES

With the `class` syntax, methods are plain functions. This means they are **not** bound. Because of this, it is common practice to bind methods in the constructor as needed. Doing so replaces the original method with a new bound copy.

```
class Baz {
  constructor() {
    this.bar = 7;
    this.foo = this.foo.bind(this); // boundValue = the instance
  }

  foo() { // context = ?
    return this.bar
  }
}
```

WHEN TO BIND METHODS

The short answer is "when the function is passed around."

BINDING UNNECESSARY

```
class Baz {
  constructor() {
    this.bar = 7;
    this.foo = this.foo.bind(this); // boundValue = the instance
  }

  foo() { // context = ?
    return this.bar
  }

  doubleFoo() {
    // context = the instance; returns (2 * 7)
    return 2 * this.foo();
  }
}
```

BINDING NECESSARY

```
class Baz {
  constructor() {
    this.bar = 7;
    this.doubleFoo = this.doubleFoo.bind(this);
    // boundValue = the instance
  }

  foo() { // context = ?
    return this.bar
  }

  doubleFoo() {
    // context = the instance; returns (2 * 7)
    return 2 * this.foo();
  }

  doubleFooGetter() {
    return this.doubleFoo; // boundValue = the instance
```

JSX

Props and attributes in JSX get passed to the `React.createElement` function, meaning they need to be bound.

```
import React, { Component } from 'react';

class Toggle extends Component {
  constructor(props) {
    super(props);
    this.state = { enabled: false };
    this.toggle = this.toggle.bind(this); // boundValue = the instance
  }

  toggle() {
    this.setState({ enabled: !this.state.enabled });
  }

  render() {
    return (
      <button onclick="{this.toggle}">
        { this.state.enabled ? 'Enabled' : 'Disabled' }
      </button>
```

CONCLUSION

Allowing `this` to be variable based on context is confusing and error prone

In modern JS, following a few guidelines makes `this` much easier to deal with.

- Always arrow functions instead of plain `function` declarations
- Bind instance methods in the constructor if they are passed around