The Complete Rules to `this`

Master 'this' by learning exactly how its rules work in JavaScript. Eliminate the cloud of uncertainty surrounding this keyword. Learn exactly how 'this' is determined, how to predict what it will be before you run your code, and how to leverage its uses.

We've gone over several different ways that `this` is set inside functions. We still haven't covered the exact rules that JavaScript is using to do it. This lesson will make you a master of `this`.

Rules

1 - If the `new` keyword is used when calling the function, `this` inside the function is a brand new object created by the JavaScript engine.

```
function ConstructorExample() {
    console.log(this);
    this.value = 10;
    console.log(this);
}

new ConstructorExample();

// -> ConstructorExample {}
// -> ConstructorExample { value: 10 }
```

2 - If apply, call, or bind are used to call a function, `this` inside the function is the object that is passed in as the argument.

```
function fn() {
    console.log(this);
}

var obj = {
    value: 5
};

var boundFn = fn.bind(obj);
```

```
boundFn(); // -> { value: 5 }
fn.call(obj); // -> { value: 5 }
fn.apply(obj); // -> { value: 5 }
```

3 - If a function is called as a method — that is, if dot notation is used to invoke the function — `this` is the object that the function is a property of. In other words, when a dot is to the left of a function invocation, `this` is the object to the left of the dot. (`ƒ` symbolizes function in the code blocks)

```
const obj = {
    value: 5,
    printThis: function() {
      console.log(this);
    }
};

obj.printThis(); // -> { value: 5, printThis: ƒ }
```

4 - If a function is invoked as a *free function invocation*, meaning it was invoked without any of the conditions present above, `this` is the global object. In a browser, it's `window`.

```
function fn() {
    console.log(this);
}

// If called in browser:
fn(); // -> Window {stop: ƒ, open: ƒ, alert: ƒ, ...}
```

*Note that this rule is the same as rule 3 — the difference is that a function that is *not* declared as a method automatically becomes a property of the global object, `window`. This is therefore an implicit method invocation. When we call `fn()`, it's interpreted as `window.fn()`, so `this` is `window`.

```
function fn() {
    console.log(this);
}

// In browser:
console.log(fn === window.fn); // -> true
```

5 - If multiple of the above rules apply, the rule that is higher wins and will set the `this` value.

Applying the Rules

Let's go over a code example and apply our rules. Try figuring out what `this` will be with the two different function calls.

Determining Which Rule Applies

```
const obj = {
    value: 'hi',
    printThis: function() {
        console.log(this);
    }
};

const print = obj.printThis;
obj.printThis(); // -> {value: "hi", printThis: ƒ}
print(); // -> Window {stop: ƒ, open: ƒ, alert: ƒ, ...}
```

`obj.printThis()` falls under rule 3 — invocation using dot notation. On the other hand, `print()` falls under rule 4 as a free function invocation. For `print()` we don't use `new`, `bind`/`call`/`apply`, or dot notation when we invoke it, so we go to rule 4 and `this` is the global object, `window`.

This goes back to value vs. reference. The value of `printThis` on the object is a reference to the function. When we assign `obj.printThis` to `print`, `print` receives the reference of the function. It's not bound to `obj` in any way - `obj` just happens to have a reference to it.

When we invoke it without `obj`, it's an FFI. It really is the use of the dot (`.`) that makes rule 3 apply.

When Multiple Rules Apply

When multiple rules apply, the rule higher on the list wins. If rules 2 and 3 both apply, rule 2 takes precedence.

```
const obj1 = {
    value: 'hi',
    print: function() {
        console.log(this);
        console.log(this);
    },
};

const obj2 = { value: 17 };

obj1.print.call(obj2); // -> { value: 17 }
```

If rules 1 and 3 both apply, rule 1 takes precendence.

```
const obj1 = {
    value: 'hi',
    print: function() {
        console.log(this);
    },
};

new obj1.print(); // -> print {}
```

Libraries

Libraries will sometimes intentionally bind the value of `this` inside their functions. `this` is bound to the most useful value for use in the function. jQuery, for example, binds `this` to the DOM element triggering an event in the callback to that event. If a library has an unexpected `this` value that doesn't seem to follow the rules, check its documentation. It's likely being bound using `bind`.

Arrow Functions

ES2015 arrow functions get their `this` value lexically. We'll cover this in its entirety in the next lesson.

That's it.