Arrow Functions and `this`

Learn how arrow functions interact with the `this` keyword. Learn how we can leverage their new rules of 'this' binding to make our code more intuitive and clean.

Arrow functions and `this`

There is one rule we left out of the previous lesson. Arrow functions don’t follow any of the traditional rules of `this`-binding.

Instead, arrow functions get their `this` binding from their scope. We’ve discussed five rules to `this`-binding. We can now add a 6th. Here they all are.

Rules

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

```
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)

```
var 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.

6 - **If the function is an ES2015 arrow function, it ignores all the rules above and receives the `this` value of its surrounding scope at the time it's created.** To determine what `this` is, go one line above the arrow function's creation and see what the value of `this` is there. It will be the same in the arrow function.

```
const obj = {
    value: 'abc',
    createArrowFn: function() {
        return () => console.log(this);
    }
};

const arrowFn = obj.createArrowFn();
arrowFn(); // -> { value: 'abc', createArrowFn: ƒ }
```

Going back to the 3rd rule, when we call `obj.createArrowFn()`, `this` inside `createArrowFn` will be `obj`, as we're calling it with dot notation. `obj` therefore gets bound to `this` in `arrowFn`. If we were to create an arrow function in the global scope in a browser, `this` would be `window`.

Why It's Useful

The alternate `this` binding rules make some things easier for us. Let's start with an example.

Incorrect value printed

```
const obj = {
    printVal: "Print value",
    generatePrintFn: function() {
        return function() {
            console.log(this.printVal);
        }
    },
};
```

```
const print = obj.generatePrintFn();
print(); // -> undefined
```

Using the rules of `this`, we can figure out why this is happening. The function returned to us is invoked as a free-function invocation. There's no dot and nothing bound. Therefore, `this` becomes `window` and there's no `printVal` available on `window`, so it prints `undefined`.

We could solve this problem using `apply`/`call`/`bind` which allow us to set the `this` value ourselves.

Using function.bind

```
const obj = {
    printVal: "Print value",
    generatePrintFn: function() {
        console.log(this.printVal);
    },
};

const print = obj.generatePrintFn.bind(obj);
print(); // -> Print value
```

Another solution would be to use the `var self = this;` hack.

```
const obj = {
    printVal: "Print value",
    generatePrintFn: function() {
        var self = this;

        return function print() {
            console.log(self.printVal);
        }
    },
};

const print = obj.generatePrintFn();
print(); // -> Print value
```

Here, we're taking advantage of a closure to store the correct `this` value in another variable and use it later.

() => {}

() => {}

Arrow functions provide a more elegant solution than any of these.

```
const obj = {
    printVal: "Print value",
    generatePrintFn: function() {
        return () => console.log(this.printVal);
    },
};

const print = obj.generatePrintFn();
print(); // -> Print value
```

Using an arrow function, `this` inside the new returned function is permanently set to `obj`. When `generatePrintFn` was originally called in line 8 above, `this` was `obj` due to the use of dot notation. `obj` is therefore permanently set as the `this` value of the returned function.

Obviously this is a contrived example, but you’ll find yourself coming across this issue in other places.