The 'this' Keyword & Binding in JavaScript

In this lesson, we study how the notorious 'this' keyword works in JavaScript in the context of explicit, implicit, new, and global binding.

What is *this*?

Almost all beginner JavaScript programmers struggle with the `this` keyword. The good news is that understanding `this` is actually much easier than it seems. In a nutshell, the value of `this` depends on what context it is used in. So, if it is used in a function, it’s value will depend on how that function is invoked, i.e., the call site. Let’s go through the ways that `this` can be assigned in JavaScript.

Implicit Binding

When the dot notation is used to call a function that is in an object or an object of a class, we say that `this` was bound implicitly. For example, consider the code sample below:

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  getName() {
    return `${this.firstname} ${this.lastname}`;
  }
}

var me = new Developer('Robin', 'Wieruch');
console.log(me.getName()); // 'this' is me
var hobbit = new Developer('Frodo', 'Baggins');
console.log(hobbit.getName()); // 'this' is 'hobbit'
```

Left of the Dot Rule

Whatever is to the *left of the dot* is what `this` is. For example, we call `getName()` first through the `me` object and hence `firstname` and `lastname` of the `me` object, i.e., 'Robin' Wieruch' get printed. Then we call `getName()` through the `hobbit` object and so `this` refers to that `hobbit` object and prints its attributes.

Explicit Binding

Unlike implicit binding, where the function is part of the object, and so it becomes obvious what `this` is, standalone functions can be *bound* explicitly to objects at call time.

`call()` & `apply()`

Consider the code block below. The function, `printName()` is bound explicitly to the `me` object of the `Developer` class on *line 15* using the `call()` function. Furthermore, if `printName()` is called without any object bound to it, as on *line 18* it prints the first and last names as undefined because `this` *is* undefined. Arguments can be passed to a function using `call()` as on *line 28*. However, if you do not want to pass each argument individually and instead pass all your arguments as an array, you can use the `apply()` function as on *line 31*.

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}

var printName = function() {
  console.log(`My name is ${this.firstname} ${this.lastname}`);
};

var me = new Developer('Robin', 'Wieruch');

// '.call()' can be used to explicitly bind a function to an object
printName.call(me);

// printName() is not bound to an object so 'this' is undefined
printName();
```

```
var printInfo = function(lang1, lang2, lang3) {
  console.log(`My name is ${this.firstname} ${this.lastname} and I know ${lang1}, ${lang2}, a
}


// Create an array of languages
languages = ['Javascript','C++', 'Python'];

// Pass each argument individually by indexing the array
printInfo.call(me, languages[0], languages[1], languages[2]);

// Pass all the arguments in one array to .apply()
printInfo.apply(me, languages);
```

`bind()`

When called on a function, `.bind()` sets a `this` context and returns a new function with a bound `this` context. Consider the code below.

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}

var printName = function() {
  console.log(`My name is ${this.firstname} ${this.lastname}`);
};

var me = new Developer('Robin', 'Wieruch');

// Here we bind the me object to the printName() function and get a new function called newPr
const newPrintName = printName.bind(me);

// bound newPrintName() prints appropriately
newPrintName();

// unbound printName() prints undefined
printName();
```

New Binding

`this` can be defined explicitly within a function as it can be in a class. Consider the code block below. Try experimenting with the values of `this.firstname` and `this.lastname` in the `printInfo` function and see what happens.

```
var printInfo = function(firstname, lastname, lang1, lang2, lang3) {
  this.firstname = firstname;
  this.lastname = lastname;
  console.log(`My name is ${this.firstname} ${this.lastname} and I know ${lang1}, ${lang2}, a
}

languages = ['Javascript','C++', 'Python'];
printInfo('Robin', 'Wieruch', languages[0], languages[1], languages[2]);
```

Global Context

When `this` is used outside of any context such as a class, function, or object, it refers to the *global object*. The global object in the browser is usually the `window` object. Download the file below, which simply prints the global `this` object, and open it with a browser of your choice. Then examine your browser's console (inspect element > console). It will say something like "window{document:...". That is the global window object that `this` refers to. In the case of a terminal and in our case the global object is undefined.

```
<html>
    <head>
        <script>console.log(this)</script>
    </head>
</html>
```

```
// this is undefined in terminals/command prompts
console.log(this);
```

Caveat: arrow functions and 'this'

Remember Arrow Functions? Let's see how the `this` keyword works with them. Consider the code block below. What do you think should get printed?

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }
}

// declare getName() outside of the Developer class using arrow functions
var getName = () => console.log(this.firstname);

var me = new Developer('Robin', 'Wieruch');
const printMyName = getName.bind(me);

printMyName();
```

Yep, this gives you an error. This is because methods like `apply()`, and `bind()`, etc. don't have any effect on `this` in an arrow function in Javascript. The value of `this` remains the same as it was when the function was called. If you want to bind to a different value, you need to use a function expression.

Coding Challenge: Fix the error

The following code has a bug. Can you find it and fix it?

```
let me = {
  name: "Robin",
  getName: function(){
    console.log(this.name);
  }
}

var getMyName = me.getName;
getMyName();
```

We'll look at how class inheritance in Javascript works in the next lesson.