Getting Started with ECMAScript 6 Arrow functions – The "this" keyw

Other than syntax, the biggest change that ECMAScript Arrow Functions brings about, is how it relates to context: in the function body, "this" has a different meaning than you may expect.

It seems to me that the two problems that are solved by ECMAScript 6 Arrow functions are: verbose syntax and the tricky nature of "**this**" inside a function. Granted, when working with methods and constructors, the meaning of "**this**" is a bit easier to understand. But when working with functions that are **not** constructors, or **not** bound to an object, the "**this**" keyword will resolve to the window object, and that is rarely the behavior you want. In a function that is defined in the global scope, it is probably unlikely that you will intentionally want to refer to "**this**" (and global function declarations should really be kept to a minimum or even better, avoided at all costs). But when you have a function that is declared inside of a method for example, it is not at all uncommon to attempt access to "**this**" from the nested function. This is where the frustration starts, because the meaning of "**this**" will vary depending on the circumstances.

Example # 1A

```
var foo = {
  color: "red",
  getColor: function(){
    function privateGetColor(){
      return this.color;
    };

    return privateGetColor();
  }
};

foo.getColor(); // undefined
```

In Example # 1A, the object foo has two properties: "**color**", whose value is: "**red**", and the method: "**getColor**". The method **foo.getColor** has a nested function: **privateGetColor**, which references "**this**". The problem is: inside of **privateGetColor**, "**this**" refers to the window object. Since there is no **window.color**, privateGetColor returns: "**undefined**", which means that **foo.getColor()** returns "**undefined**".

Example # 1B

```
var foo = {
  color: "red",
  getColor: function(){
    var me = this;

```

```
    function privateGetColor(){
      return me.color;
    };

    return privateGetColor();
  }
};

foo.getColor(); // "red"
```

In Example # 1B, we have fixed the situation by creating a private variable named: "**me**" inside of **foo.getColor**, which caches "**this**". This way, the nested function: "**privateGetColor**" now has access to "**this**", and this foo.getColor returns "**red**".

Example # 2

```
var foo = {
  color: "red",
  getColor: function(){
    var privateGetColor = () => this.color;

    return privateGetColor();
  }
};

foo.getColor(); // "red"
```

In Example # 2, we have a more elegant solution. By leveraging an arrow function, we no longer need to create the variable "**me**", in order to cache the value of "**this**". Now that the nested function: privateGetColor is an arrow function, we can reference "**this**" in the arrow function's body. Since privateGetColor now returns "**red**", **foo.getColor()** returns "**red**".

Lexical binding of "this"

The reason that Example # 2 saves the day, is because of the change in meaning for the "**this**" keyword inside of an arrow function. Normally, "this" will refer to the object of which the function is a method. With arrow functions, "**this**" is bound lexically. This means that it is where an arrow function is defined that affects the meaning of "**this**", not where it is executed.

Example # 3

```
var bar = {color: "blue"};

var foo = {
  color: "red",
  getColor: function(){
    var privateGetColor = () => this.color;

    return privateGetColor.call(bar);
  }
};

foo.getColor(); // "red"
```

In Example # 3, we have an object named "**bar**", which has a "**color**" property. When we execute **foo.getColor()**, we use the call method to change the context in which **getColor** is executed. This should have changed the meaning of "**this**" to "**bar**", which would result in the return value of "**blue**" *(i.e. privateGetColor.call(bar) should return: "blue")*. But that is not what happens; the return value of **foo.getColor()** is: "**red**".

The reason for all of this is that inside of an arrow function, "**this**" is bound lexically. So, it is where an arrow function is **DEFINED**, not where it is executed, that determines the meaning of "**this**". It might help of think of how scope works in JavaScript. The lexical binding of "**this**" inside the body of an arrow function behaves in a similar way. The closest containing object outside of the arrow function will be resolved as "**this**".

Summary

The meaning of "**this**" inside an arrow function is without doubt a significant change in the JavaScript specification. Since ECMAScript 6 is 100% backwards-compatible, this has no effect on the meaning of "**this**" inside of normal function definitions, function expressions, constructors or methods. While it may take a while to get used to this concept, the ramifications are very positive. The ability to reference a specific object inside of a click-handler or AJAX calls makes for code that will be easier to read, manage and extend.

One Comment

What Are the Best Links for Learning About ECMAScript 6 ? | Kevin Chisholm - Blog

[...] http://blog.kevinchisholm.com/javascript/ecmascript-6/getting-started-with-ecmascript-6-arrow-functi… [...]

November 27, 2015 at 3:25 pm

Comments are closed.