When You Should Not Use Arrow Functions

Summary: in this tutorial, you will learn when you **should not use** the arrow functions in ES6.

An arrow function doesn't have its own `this` value and the `arguments` object. Therefore, you should not use it as an event handler, a method of an object literal, a prototype method, or when you have a function that uses the `arguments` object.

1) Event handlers

Suppose that you have the following input text field:

```
<input type="text" name="username" id="username" placeholder="Enter a username">
```

And you want to show a greeting message when users type their usernames. The following shows the `<div>` element that will display the greeting message:

```
<div id="greeting"></div>
```

Once users type their usernames, you capture the current value of the input and update it to the `<div>` element:

```
const greeting = document.querySelector('#greeting');
const username = document.querySelector('#username');
username.addEventListener('keyup', () => {
  greeting.textContent = 'Hello ' + this.value;
});
```

However, when you execute the code, you will get the following message regardless of whatever you type:

```
Hello undefined
```

It means that the `this.value` in the event handler always returns `undefined`.

As mentioned earlier, the arrow function doesn't have its own `this` value. It uses the `this` value of the enclosing lexical scope. In the above example, the `this` in arrow function references the global object.

In the web browser, the global object is `window`. The `window` object doesn't have the `value` property. Therefore, the JavaScript engine adds the value property to the `window` object and sets its values to `undefined`.

To fix this issue, you need to use a regular function instead. The `this` value will be bound to the `<input>` element that triggers the event.

```
username.addEventListener('keyup', function () {
    input.textContent = 'Hello ' + this.value;
});
```

2) Object methods

See the following `counter` object:

```
const counter = {
  count: 0,
  next: () => ++this.count,
  current: () => this.count
};
```

The `counter` object has two methods: `current()` and `next()`. The `current()` method returns the current counter value and the `next()` method returns the next counter value.

The following shows the next counter value which should be 1:

```
console.log(counter.next());
```

However, it returns `NaN`.

The reason is that when you use the arrow function inside the object, it inherits the `this` value from the enclosing lexical scope which is the global scope in this example.

The `this.count` inside the `next()` method is equivalent to the `window.count` (in the web browser).

The `window.count` is `undefined` by default because the `window` object doesn't have the `count` property. The `next()` method adds one to `undefined` that results in `NaN`.

To fix this, you use regular functions as the method of an object literal as follows:

```
const counter = {
    count: 0,
    next() {
        return ++this.count;
    },
    current() {
        return this.count;
    }
};
```

Now, calling the `next()` method will return one as expected:

```
console.log(counter.next()); // 1
```

3) Prototype methods

See the following `Counter` object that uses the `prototype` pattern:

```
function Counter() {
    this.count = 0;
}

Counter.prototype.next = () => {
    return this.count;
```

```
};

Counter.prototype.current = () => {
    return ++this.next;
}
```

The `this` value in these `next()` and `current()` methods reference the global object. Since you want the `this` value inside the methods to reference the `Counter` object, you need to use the regular functions instead:

```
function Counter() {
    this.count = 0;
}

Counter.prototype.next = function () {
    return this.count;
};

Counter.prototype.current = function () {
    return ++this.next;
}
```

4) Functions that use the arguments object

Arrow functions don't have the `arguments` object. Therefore, if you have a function that uses `arguments` object, you cannot use the arrow function.

For example, the following `concat()` function won't work:

```
const concat = (separator) => {
    let args = Array.prototype.slice.call(arguments, 1);
    return args.join(separator);
}
```

Instead, you use a regular function like this:

```
function concat(separator) {
    let args = Array.prototype.slice.call(arguments, 1);
    return args.join(separator);
}
```

Summary

- An arrow function doesn't have its own `this` value. Instead, it uses the `this` value of the enclosing lexical scope. An arrow function also doesn't have the `arguments` object.
- Avoid using the arrow function for event handlers, object methods, prototype methods, and functions that use the `arguments` object.