Context and Explicit Setting

(https://github.com/learn-co-curriculum/phase-1-context-explicit-setting) (https://github.com/learn-co-curriculum/phase-1-context-explicit-setting/issues/new)

Learning Goals

- Explicitly override the default context with `call` and `apply`
- Explicitly lock the context object for a function with `bind`

Introduction

In the previous lesson, we learned that when we invoke functions, JavaScript creates a function execution context that includes a context object (`this`) which is made available to the function. That context object will be set to "whatever's to the left of the dot" or, when there's nothing to the left of the dot, the global object.

JavaScript provides other means for specifying what *we* want the context object to be. These are the *explicit* methods of setting the context object: `call` , `apply` , and `bind` .

Explicitly Override Context with `call` and `apply`

Recall that `call` and `apply` are *methods on functions* that provide an alternate way to call the function and enable us to override the default context object.

Let's think back to a previous lesson and recall working with records. We'll invoke the functions in a familiar way, but also show how we could achieve the equivalent behavior using `call` or `apply` .

```
const asgardianBrothers = [
  {
    firstName: "Thor",
    familyName: "Odinsson"
```

```
  },
  {
    firstName: "Loki",
    familyName: "Laufeysson-Odinsson"
  }
]

function intro(person, line) {
  return `${person.firstName} ${person.familyName} says: ${line}`
}

const phrase = "I like this brown drink very much, bring me another!"
intro(asgardianBrothers[0], phrase) //=> Thor Odinsson says: I like this brown drink very much, bring me another!
```

When we first wrote a record-oriented program, we wrote functions in the style of `intro`. They took the record *as an argument*. In fact, if we look at the `solution` branch for the previous lesson, we'll see that multiple functions have the same first parameter:

```
function createTimeInEvent(employee, dateStamp){ /* */ }
function createTimeOutEvent(employee, dateStamp){ /* */ }
function hoursWorkedOnDate(employee, soughtDate){ /* */ }
```

Your solution probably has a similar repetition.

What if we told JavaScript that instead of the record being a *parameter*, it could be assumed as a *context* and thus accessible via `this`. To accomplish this, we can use either `call` or `apply`:

```
function introWithContext(line){
  return `${this.firstName} ${this.familyName} says: ${line}`
}

introWithContext.call(asgardianBrothers[0], phrase)
//=> Thor Odinsson says: I like this brown drink very much, bring me another!
```

```
introWithContext.apply(asgardianBrothers[0], [phrase])
//=> Thor Odinsson says: I like this brown drink very much, bring me another!
```

Note that, unlike the `intro` function, `introWithContext` expects only a catchphrase as an argument. We can then call either `call` or `apply` on `introWithContext`, passing a `thisArg` object as the first argument; that argument becomes the `this` *inside* the function. (See the documentation for `call` and `apply` for further clarification.)

As seen above, the function calls using `call` or `apply` are nearly identical: the only difference is in how additional arguments (if any) are passed to them. In the case of `call`, which can take any number of arguments, the additional arguments are listed individually after the `thisArg`. They are passed along to the function that `call` is being called on the same way that arguments inside of a `()` are passed. Additional arguments to `apply`, on the other hand, are passed inside an array. When `apply` is called, the array of arguments gets destructured and the arguments it contains are passed along just as they are with `call`.

All three approaches for calling our introduction function yield the same results:

```
intro(asgardianBrothers[0], phrase) === introWithContext.call(asgardianBrothers[0], phrase) //=> true
intro(asgardianBrothers[0], phrase) === introWithContext.apply(asgardianBrothers[0], [phrase]) //=> true

const complaint = "I was falling for thirty minutes!"
intro(asgardianBrothers[1], complaint) === introWithContext.call(asgardianBrothers[1], complaint) //=> true
intro(asgardianBrothers[1], complaint) === introWithContext.apply(asgardianBrothers[1], [complaint]) //=> true
```

Explicitly Lock Context For a Function With `bind`

Let's suppose that we wanted to create a copy of the `introWithContext` function in which the context is permanently bound to `asgardianBrothers[0]`. This is where `bind` comes in:

```
const asgardianBrothers = [
  {
    firstName: "Thor",
    familyName: "Odinsson"
```

```
  },
  {
    firstName: "Loki",
    familyName: "Laufeysson-Odinsson"
  }
]
function introWithContext(line){
  return `${this.firstName} ${this.familyName} says: ${line}`
}

const thorIntro = introWithContext.bind(asgardianBrothers[0])
thorIntro("Hi, Jane") //=> Thor Odinsson says: Hi, Jane
thorIntro("I love snakes") //=> Thor Odinsson says: I love snakes
```

We call the `bind` method on a function (`introWithContext` in this case) and pass one argument: the object we want to be bound to the function. The `bind` method ***returns a new function that needs to be called***, which we've saved into the variable `thorIntro` . Then, when we call `thorIntro` , any `this` references inside the original function that `bind` was called on are "hard set" to `asgardianBrothers[0]` .

Conclusion

To sum up the explicit overrides:

1. The context object can be explicitly set in a function by invoking `call` on the function and passing an object (`thisArg`) as the first argument; the object can then be accessed via `this` inside the function. Additional parameters to the function are listed after `thisArg` .
2. The context object can be explicitly set in a function by invoking `apply` on the function and passing an object (`thisArg`) as the first argument; the object can then be accessed via `this` inside the function. Additional parameters to the function are stored in an array which is passed as the second argument.
3. The context object can be locked in a function by invoking `bind` on the function and passing it a `thisArg` . The `bind` function makes a copy of the functionality of the function it was called on, but with all the `this` stuff locked in place, and returns that function. That *new* function can have arguments passed to it with `()` as usual.

Resources

- `call` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/call)
- `apply` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/apply)
- `bind` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind)