Context and Implicit Setting

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

Learning Goals

- Learn how JavaScript implicitly sets execution context

Introduction

In the previous lesson we provided definitions of:

- Execution Context
- `this`
- `call`
- `apply`
- `bind`

Recall that, when a function in JavaScript ***is called***, a *function execution context* is created. This execution context includes an object that is passed to the function at the time it's called, either implicitly or explicitly. This object can be referred to from inside the function using the `this` keyword.

We *explicitly* pass the object by using the `call`, `apply`, or `bind` method and passing the desired context object as an argument. We will learn the explicit approach in the next lesson.

But what if we call a function *without* using one of those methods — how does JavaScript know what object to pass to the function? JavaScript uses a couple of simple rules to determine what object should be passed along, depending on the circumstances under which a function is called. We will learn how this works in this lesson.

> **Top Tip**: as always, it's important to follow along with the examples we'll be presenting. We encourage you to get familiar with two different programming environments ("runtimes") that allow you to try out JavaScript code: 1) the browser console, and 2) the NodeJS environment,

> which you can launch by running `node` in your terminal. Play around with both to see which you prefer! (**Note:** your output may differ from what's shown in this lesson depending on which runtime you use.)

How JavaScript Implicitly Sets Execution Context

Under most circumstances, we do not need to use `call`, `apply`, or `bind` to explicitly pass the `this` object. Instead we depend on JavaScript to determine what it should be *implicitly*. As a result, it's important to understand how that works. There are two basic conditions we need to consider:

1. "bare" function calls
2. calling a function expression that's a property of an object

We'll start with the second case.

Functions as Properties of Objects

In the lessons on JavaScript objects, we learned that object properties can be *primitive* values (numbers, strings, etc.) **or** data structures (arrays or objects). But object properties can also be **functions**. Consider this example:

```
const byronPoodle = {
  name: "Byron",
  ageInYears: 2,
  warn: function() {
    console.log(`Bark bark bark`);
  }
};
```

The `name` and `ageInYears` keys point to primitive values, and the `warn` key points to a *function expression*. We can access any of the properties using dot notation:

```
byronPoodle.name;
// LOG: Byron
```

```
byronPoodle.ageInYears;
// LOG: 2
byronPoodle.warn;
// LOG: ƒ () {
//    console.log(`Bark bark bark`);
// }
```

To call the `warn` function, we would simply append the `()`:

```
byronPoodle.warn();
// LOG: Bark bark bark
```

Note: When functions are defined as function expressions inside an object, they are often referred to as `method`s. We've seen plenty of methods already. For example, in earlier lessons we learned how to use `Array` methods. Any time we create an array, that array *inherits* all the properties that belong to `Array` objects in JavaScript, including all of their methods. We can call those array methods in exactly the same way we call `byronPoodle`'s `warn` method:

```
const arr = [1,2,3];  // the created array inherits all array methods
arr.pop();            // so we can call those methods using dot notation
// => 3
arr;
// => [1,2]
```

The Execution Context of Methods

When you call a function expression that's a property of an object — a method — that function expression's `this` object is quite simply the object on which the function is called, i.e., the object to the left of the dot. Recall from the previous lesson that the `this` keyword can be used inside a function to access the object and its properties; let's log `this` from inside the `warn` method:

```
const byronPoodle = {
  name: "Byron",
  ageInYears: 2,
```

```
  warn: function() {
    console.log(`Bark bark bark`);
    console.log(this);
  }
};
```

and call our method:

```
byronPoodle.warn();
// LOG: Bark bark bark
// LOG: {name: "Byron", ageInYears: 2, warn: f}
```

As you can see, the value of `this` in `byronPoodle`'s `warn` method is the object itself. This makes sense logically: when we call a method (i.e., a function that belongs to an object), the context for that method is the object it belongs to.

What this means is that we can also use `this` in much more sophisticated ways inside our methods. Let's create a new object, `blakeDoodle`:

```
const blakeDoodle = {
  name: "Blake",
  breed: "Labradoodle",
  sonicAttack: "ear-rupturing atomic bark",
  mostHatedThing: "noises in the apartment hallway",
  warn: function() {
    console.log(`${this.name} the ${this.breed} issues an ${this.sonicAttack} when he hears ${this.mostHatedThing}`);
  }
};

blakeDoodle.warn();
// LOG: Blake the Labradoodle issues an ear-rupturing atomic bark when he hears noises in the apartment hallway
```

To summarize, any time you call `someObject.someFunction()`, the `this` object inside of `someFunction` will be the thing to the left of the `.`: `someObject`.

Let's look at another example. First, we'll create a couple of objects and a function expression that we'll assign to the variable name `speak`:

```
const frog = { name: "Kermit" };
const pig = { name: "Miss Piggy" };
const speak = function() { return `It ain't easy being ${this.name}`};
```

We can then assign the `speak` function as a property to each of our objects:

```
frog.speak = speak;
pig.speak = speak;
frog.speak === pig.speak; //=> true
```

We see above that both `pig` and `frog` have the **same** `speak` property, but let's see what happens when we call the method on each of our objects:

```
frog.speak();  //=> "It ain't easy being Kermit"
pig.speak();  //=> "It ain't easy being Miss Piggy"
```

The context used *inside* the function `speak` is defined by what's "left of the dot."

The Execution Context of "Bare" Function Calls

What happens if we invoke a function that's **not** defined inside an `Object`:

```
const contextReturner = function() {
  return this;
}

contextReturner(); //=> window
contextReturner() === window; //=> true
```

When no object is to the left of the function, JavaScript invisibly adds **the global object**. A simple way of saying it: when you call `someFunction()`, the context inside of `someFunction` will be the thing to the left of the `.`. Since there's nothing there, JavaScript swaps in the global object.

In the browser-based runtime, the global object is called `window`. In NodeJS, it's called `global`.

Let's look at an example in Chrome:

```
const locationReturner = function() {
  return this.location.host;
}

locationReturner(); //=> URL host serving the current page e.g. developer.mozilla.org
```

It's worth noting that even in a function inside of another function, the inner function's default context is still the global object:

```
function a() {
  return function b() {
    return this;
  }
}

a()() === window; //=> true
```

Prevent Implicitly Setting a Global Object In Function Calls With `use strict`

We wish we could say that the default context was **always** the global object. It'd make things simple.

However, in JavaScript, if the engine sees the `String` "use strict" inside a function, it will *stop* passing the global object. If JavaScript sees `"use strict"` at the top of a JavaScript code file, it will apply this rule (and other strict behaviors) to *all functions*.

```
function looseyGoosey() {
  return this;
}

function noInferringAllowed() {
  "use strict";
  return this;
}

looseyGoosey() === window; //=> true
noInferringAllowed() === undefined; //=> true
```

There are really no guidelines as to which you'll see more. Some programmers think `strict` prevents confusing bugs (seems wise!); others think it's an obvious rule of the language and squelching it is against the language's love of functions (a decent argument!). Generally, we advise you to think of the "default mode" as the one that permits an *implicit* presumption of context. For more on strict mode, see the Resources.

Special Case: Implicitly-Set Context in Object-Oriented Programming

This lesson covers how `this` is implicitly set. An important place where this happens is when new instances of classes are created. Class definition and instance creation are hallmarks of a programming approach known as `object-oriented ("OO") programming`. A `class` describes a group or category of some sort (e.g., poodles) and specifies the characteristics (properties) that all instances of the class (specific poodles) are expected to have. The specific instances are created using a function known as a `constructor` that we define inside our `Poodle` class. It might look something like this:

```
class Poodle{
  constructor(name, pronoun){
    this.name = name;
    this.pronoun = pronoun;
    this.sonicAttack = "ear-rupturing atomic bark";
    this.mostHatedThing = "noises in the apartment hallway";
  }
```

```
  warn() {
    console.log(`${this.name} issues an ${this.sonicAttack} when ${this.pronoun} hears ${this.mostHatedThing}`);
  }
}
```

All new members of the class (instances) will have all of the properties that are defined in the constructor as well as any methods defined inside the Class. For our example, when we create a new instance, that new object is the implicitly-set context object for the class's `warn` method:

```
const byron = new Poodle("Byron", "he");
byron.warn(); //=> Byron issues an ear-rupturing atomic bark when he hears noises in the apartment hallway
```

You do not need to understand the details of the OO Programming paradigm. The important point here is that JavaScript's rules about implicitly setting execution context extend naturally to the OOP case.

Conclusion

To sum up the discussion thus far:

1. Execution context is set at function call-time and includes a context object (`this`) that is passed to the function, either implicitly or explicitly.
2. In "bare" function calls, the context object is automatically set to the global object unless prevented by `"use strict"` .
3. In "non-bare" function calls, the context object is automatically set to the "object to the left of the dot."
4. (For Object-Oriented JavaScript) The context object defaults to the new thing being created in a `class` 's `constructor`

This covers the *implicit* context-setting rules. We'll now learn about the *explicit* context-setting rules.

Resources

- **strict (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode)**