The Lost Context Bug Code-Along

(https://github.com/learn-co-curriculum/phase-1-the-lost-context-bug) (https://github.com/learn-co-curriculum/phase-1-the-lost-context-bug/issues/new)

Learning Goals

- State the cause of the lost context bug
- Use a `thisArg` to avoid the lost context bug
- Use a closure to regain access to the lost context
- Use an arrow function expression to create a function without its own context

Introduction

In the previous lessons we've learned about record-oriented programming and how, by using methods like `call`, `apply`, and `bind`, we can change the default context of a function from the global context (`window` in the browser, `global` in NodeJS) as we see fit. That's an awesome power.

However, sometimes the rules of function execution interact in a way that leads to ***one particularly surprising bug***: "the lost context bug." It's impossible to list *all* the places where this bug could be triggered, but if you encounter something "strange" like what we describe below, you'll know how to proceed.

Scenario

To follow along, open the `index.html` file in your browser and open the console. You will add your code in `index.js`.

It's the All-Father Odin's birthday. His sons, Thor and Loki, would like to print him a birthday greeting using JavaScript. They know how to define `Object`s and `function`s, so they've written a simple function that takes an `Object` (`messageConfig`) as context and prints a JavaScript greeting card.

The `Object` looks like this:

```
const messageConfig = {
  frontContent: "Happy Birthday, Odin One-Eye!",
  insideContent:
    "From Asgard to Nifelheim, you're the best all-father ever.\n\nLove,",
  closing: {
    Thor: "Admiration, respect, and love",
    Loki: "Your son",
  },
  signatories: ["Thor", "Loki"],
};
```

To display this, they wrote the following function:

```
const printCard = function () {
  console.log(this.frontContent);
  console.log(this.insideContent);

  this.signatories.forEach(function (signatory) {
    const message = `${this.closing[signatory]}, ${signatory}`;
    console.log(message);
  });
};

printCard.call(messageConfig);
```

This doesn't work as planned. They get an error like the following:

```
Happy Birthday, Odin One-Eye!
From Asgard to Nifelheim, you're the best all-father ever.

Love,
/Users/heimdall/git_checkouts/fi/jscontext/unnamed/card.js:20
```

```
Uncaught TypeError: Cannot read property 'Thor' of undefined
    at index.js:20
    at Array.forEach (<anonymous>)
    at Object.printCard (index.js:19)
    at index.js:25
```

What is going on here? A quick debug in the console shows that there **very much** is a property called `"Thor"` in `messageConfig.closing`:

```
console.log(messageConfig.closing.Thor); //=> "Admiration, respect, and love"
```

Here is one of the most mind-boggling problems in JavaScript: a bug created in the shadow of the all-too-easy-to-forget fact that function expressions and declarations ***inside*** of other functions ***do not automatically*** use the same context as the outer function. Think about the rules of implicit context assignment before reading on.

Debugging: Discovering the Nature of the Lost Context Bug

As a first step in getting this code working, let's add some `console.log()` calls so we can see what `this` is. Let's update our `printCard` function as follows:

```
const printCard = function () {
  console.log(this.frontContent);
  console.log(this.insideContent);

  console.log("Debug Before forEach: " + this);
  this.signatories.forEach(function (signatory) {
    console.log("Debug Inside: " + this);
    // const message = `${this.closing[signatory]}, ${signatory}`
    // console.log(message)
  });
};
```

```
printCard.call(messageConfig);
```

When we refresh the browser, this produces:

```
Happy Birthday, Odin One-Eye!
From Asgard to Nifelheim, you're the best all-father ever.

Love,
Debug Before forEach: [object Object]
Debug Inside: [object Window]
Debug Inside: [object Window]
```

The `console.log()` statements reveal the bug. *Inside* the `forEach`, the execution context **is not** the `messageConfig` `Object` we used as a `thisArg` argument when calling the function `printCard`. Instead, the `this` *inside* the function expression passed to `forEach` is the global object (`window` or `global`).

Remember the rules of function invocation. A function defaults to the *global execution context* when it is called without "anything to the left of the dot". It **does not** get its parent function's context automatically. There are many ways for programmers to solve this problem. The three most common are:

1. Explicitly specify the context by either passing a `thisArg` or using `bind`
2. Use a closure
3. Use an arrow function expression

Solution 1: Use a `thisArg` to avoid the lost context bug

Per the **forEach documentation (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)**, we could pass a `thisArg` argument to `forEach` as its second argument, after the function expression. This explicitly provides a context for the function used inside `forEach`. Doing so fixes our bug.

> **ASIDE**: This pattern works for `forEach` as well as `map` and other collection-processing methods. Consult their documentation to see where a `thisArg` is expected.

```
const printCard = function () {
  console.log(this.frontContent);
  console.log(this.insideContent);

  this.signatories.forEach(function (signatory) {
    const message = `${this.closing[signatory]}, ${signatory}`;
    console.log(message);
  }, this);
};

printCard.call(messageConfig);

/*
Happy Birthday, Odin One-Eye!
From Asgard to Nifelheim, you're the best all-father ever.

Love,
Admiration, respect, and love, Thor
Your son, Loki
*/
```

In the call to `forEach`, we tell it to use (as its own context) the context that `printCard` has as `printCard`'s `this`.

A slight variation on this idea would be to invoke `bind` on the function expression in the `forEach`:

```
const printCard = function () {
  console.log(this.frontContent);
  console.log(this.insideContent);
  const contextBoundForEachExpr = function (signatory) {
    const message = `${this.closing[signatory]}, ${signatory}`;
    console.log(message);
  }.bind(this);
```

```
  this.signatories.forEach(contextBoundForEachExpr);
};

printCard.call(messageConfig);
/*
Happy Birthday, Odin One-Eye!
From Asgard to Nifelheim, you're the best all-father ever.

Love,
Admiration, respect, and love, Thor
Your son, Loki
*/
```

In the "Context Lab" we used this approach to make sure that the reduce function in `allWagesFor` worked. Take a look at the implementation and see how `bind` -ing `reduce` saved you from falling into this bug *and* let you use the powerful `reduce` method.

Solution 2: Use a Closure to Regain Access to the Lost Context

In the previous section, we fixed the bug by taking the `this` that `printCard` has access to and either re-passing it as a `thisArg` to `forEach` **or** providing it as the context for `bind` . Alternatively, we could assign that value to a variable and leverage function-level scope and *closures* to regain access to the outer context.

```
const printCard = function () {
  console.log(this.frontContent);
  console.log(this.insideContent);

  const outerContext = this;

  this.signatories.forEach(function (signatory) {
    const message = `${outerContext.closing[signatory]}, ${signatory}`;
    console.log(message);
  });
```

```
};

printCard.call(messageConfig);
/*
Happy Birthday, Odin One-Eye!
From Asgard to Nifelheim, you're the best all-father ever.

Love,
Admiration, respect, and love, Thor
Your son, Loki
*/
```

Many JavaScript developers define the variable we called `outerContext` by the name `self`. In any case, by assigning it to a variable, we put the original context within the function-level scope that the inner function encloses as a closure. This means inside the inner function, we can get "back" to the outer function's context. That's solution number two.

What we would *really* like is for there to be a way to tell the `function` inside of `forEach` to

1. *Not* declare its own context **but also**
2. *Not* require us to do the extra work of using `bind` or a `thisArg`.

In ES6, JavaScript gave us an answer: the "arrow function expression." This is our third and most-preferred option. Nevertheless, you will see all the other approaches used in framework code (e.g. React) and in other codebases.

Solution 3: Use an Arrow Function Expression to Create a Function Without Its Own Context

As we learned earlier in this Phase, the arrow function expression (often simply called an "arrow function") is yet another way of writing a function expression. They look different from "old style" function expressions, but the ***most important difference*** is that the arrow function is ***automatically bound*** to its parent's context and does not create a context of its own.

Many programmers think arrow functions are much more predictable since they do not create their own `this` during execution and instead "absorb" the context of their enclosing environment.

Since *the whole point* of an arrow function is to ***not have its own execution context***, we should not use `call`, `bind`, or `apply` when executing them. Most of the time, you'll see them used like anonymous functions passed as first-class data into another function. See the `reduce` example below. It's typical.

You will recall that an arrow function looks like this:

```
// The `const greeter` is merely the assignment, the expression begins after the `=`
const greeter = (nameToGreet) => {
  const message = `Good morning ${nameToGreet}`;
  console.log(message);
  return "Greeted: " + nameToGreet;
};
const result = greeter("Max"); //=> "Greeted: Max"
```

Which gives the exact same result as:

```
const greeter = function (nameToGreet) {
  const message = `Good morning ${nameToGreet}`;
  console.log(message);
  return "Greeted: " + nameToGreet;
}.bind(this);
const result = greeter("Max Again"); //=> "Greeted: Max Again"
```

Because arrow functions are *so often used* to take a value, do a single operation with it, and return the result, they have two shortcuts:

- If you pass only one argument, you don't have to wrap the single parameter in `()`
- If there is only one expression, you don't need to wrap it in `{}` and the result of that expression is automatically returned.
- Anti-Shortcut: If you *DO* use `{}`, you must explicitly `return` the return value

Thus Thor and Loki can fix their problem and wish their father a happy birthday most elegantly with the following code:

```
const printCard = function () {
  console.log(this.frontContent);
  console.log(this.insideContent);
  // Wow! Elegant! And notice the arrow function's `this` is the same `this`
  // that printCard has; specifically, the `thisArg` that was passed to it
  this.signatories.forEach((signatory) =>
    console.log(`${this.closing[signatory]}, ${signatory}`)
  );
};

printCard.call(messageConfig);
/* OUTPUT:
Happy Birthday, Odin One-Eye!
From Asgard to Nifelheim, you're the best all-father ever.

Love,
Admiration, respect, and love, Thor
Your son, Loki
*/
```

Conclusion

You've now learned how to both spot and counteract the lost context bug using some very interesting tools. We think of this as a way to help protect you as you start to build your own applications.

The arrow function expression that we used here is a very important piece of syntax. While it lets us type less, and *yes* that is a very good thing, its most important feature is that ***it carries its parent's context as its own***.

With this knowledge, we think you've learned all the skills you're going to need in order to build your own JavaScript library. Enjoy the challenge!

Resources

- **foreach (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach)**
- **Arrow Function (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions)**
- **MDN On Why Arrow Functions Help Us leverage `this` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#No_separate_this)**