Introduction to Scope

 (https://github.com/learn-co-curriculum/phase-1-scope-readme)  (https://github.com/learn-co-curriculum/phase-1-scope-readme/issues/new)

Learning Goals

- Explain in general terms what the execution context is.
- Describe the difference between global- and function-scoped code.
- Understand how block scoping affects variables declared with `let` and `const`.

Introduction

Scope is, in short, the concept of **where something is available**. In the case of JavaScript, it has to do with where declared variables and methods are available within our code.

Scope is a ubiquitous concept in programming and one of the most misunderstood principles in JavaScript, frustrating even seasoned engineers. Not understanding how scope works will lead to pain.

Let's talk about Slack, baby

As the newest engineer at Flatbook, you have access to the company's Slack team. The Slack team is organized into channels, some of which are company-wide, such as the main `#general` channel, and some of which are used by individual teams for intra-team communication, such as `#education`, `#engineering`, and `#marketing`.

Each channel forms its own *scope*, meaning that its messages are only visible to those with access to the channel. Within `#engineering`, you can interact with the other members of your team, referring and responding to messages that they send. However, you can't see any of the messages posted inside `#marketing` — that's a different scope that you don't have access to.

The same exact principle of distinct scopes exists in JavaScript, and it has to do with where declared variables and functions are visible.

Execution contexts

Just as every message on Slack is sent in a channel, every piece of JavaScript code is run in an *execution context*. In a Slack channel, we have access to all of the messages sent in that channel; we can send a message that references any of the other messages posted in the same channel. Similarly, in a JavaScript execution context, we have access to all of the variables and functions declared in that context. Within an execution context, we can write an expression that references a variable or invokes a function declared in the same context.

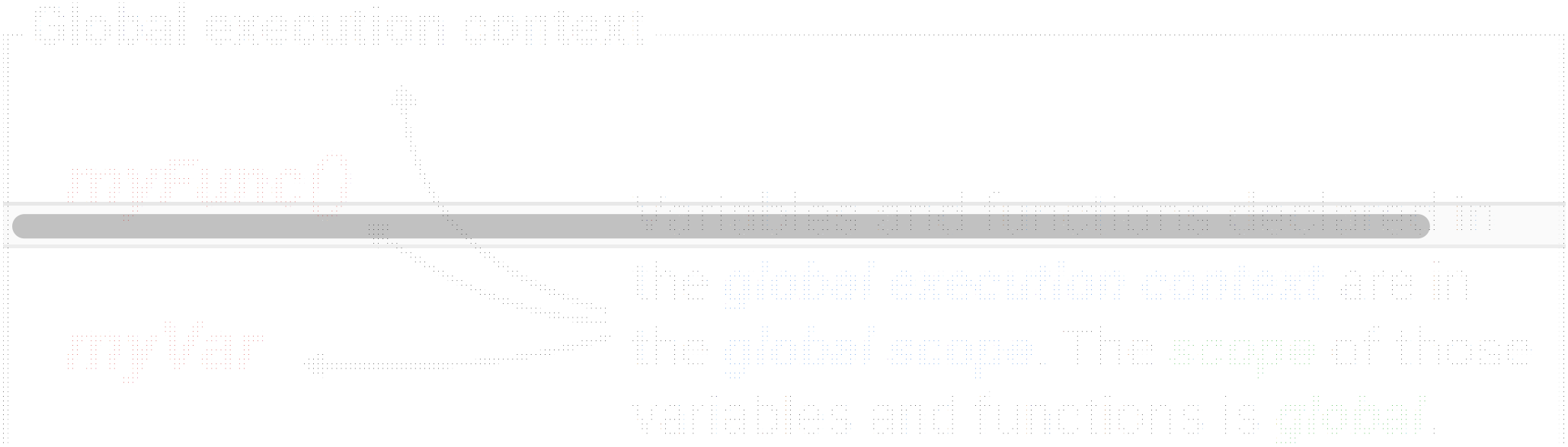
Up to this point, much of the JavaScript code we've written has lived in the *global execution context*, the context that implicitly wraps all of the JavaScript code in a project. Variables and functions declared in the global execution context — in the *global scope* — are accessible everywhere in your JavaScript code:

```
// 'myFunc' is declared in the global scope and available everywhere in your code:
function myFunc() {
  return 42;
}
// => undefined

// 'myVar' is able to reference and invoke 'myFunc' because both are declared in the same scope (the global execution
```

```
const myVar = myFunc() * 2;
// => undefined

myVar;
```



Top Tip: If a variable or function is **not** declared inside a function or block, it's in the global execution context.

Function scope

When we declare a new function and write some code in the function body, we're no longer in the global execution context. The function creates a new execution context with its own scope. Inside the function body, we can reference variables and functions declared in the function's scope:

```
function myFunc() {
  const myVar = 42;
```

```
  return myVar * 2;
}
// => undefined

myFunc();
// => 84
```

However, from outside the function, we can't reference anything declared inside of it:

```
function myFunc() {
  const myVar = 42;
}
// => undefined

myVar * 2;
// Uncaught ReferenceError: myVar is not defined
```

The function body creates its own scope. It's like a separate channel on Slack — only the members of that channel can read the messages sent in it.

Block scope

A block statement also creates its own scope... kind of.

Variables declared with `var` are **not** block-scoped:

```
if (true) {
  var myVar = 42;
}

myVar;
// => 42
```

However, variables declared with `const` and `let` **are** block-scoped:

```
if (true) {
  const myVar = 42;

  let myOtherVar = 9001;
}

myVar;
// Uncaught ReferenceError: myVar is not defined

myOtherVar;
// Uncaught ReferenceError: myOtherVar is not defined
```

This is yet another reason to **never use** `var`. As long as you stick to declaring variables with `const` and `let`, what happens in block stays in block.

The global gotcha

In a perfect world, you'd always remember to declare new variables with `const` and `let`, and you'd never run into any weird scoping issues. However, it's inevitable that at some point you're going to forget the `const` or `let` and accidentally do something like:

```
firstName = "Ada";
```

Variables created without a `const`, `let`, or `var` keyword are **always globally-scoped**, regardless of where they sit in your code. If you create one inside of a block, it's still available globally:

```
if (true) {
  lastName = "Lovelace";
}
```

```
lastName;
// => "Lovelace"
```

If you create one inside of a function — wait for it — it's still available globally:

```
function bankAccount() {
  secretPassword = "il0v3pupp135";

  return "bankAccount() function invoked!";
}

bankAccount();
// => "bankAccount() function invoked!"

secretPassword;
// => "il0v3pupp135"
```

Oh no; our super secret password has leaked into the global scope and is available everywhere! Declaring global variables and functions should only be used as a last resort if you absolutely need access to something **everywhere** in your program. In general, it's best practice to make variables and functions available only where they're needed — and nowhere else. Making a variable available in places that shouldn't have access to it can only lead to bad things and make your debugging process more complex. The more pieces of code that can access a given variable, the more places you have to check for bugs if/when that variable contains an unexpected value.

Conclusion

So, to sum up our tricks for taming the scope monster:

1. Always use `const` and `let` to declare variables.
2. Keep in mind that every function creates its own scope, and any variables or functions you declare inside of the function will not be available outside of it.
3. For Dijkstra's sake, ***always use `const` and `let` to declare variables***.

Resources

- MDN
  - **Scope (https://developer.mozilla.org/en-US/docs/Glossary/scope)**
  - **Functions — Function scope (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#Function_scope)**