You Don't Know JS Yet: Scope & Closures - 2nd Edition

Chapter 4: Around the Global Scope

Chapter 3 mentioned the "global scope" several times, but you may still be wondering why a program's outermost scope is all that important in modern JS. The vast majority of work is now done inside of functions and modules rather than globally.

Is it good enough to just assert, "Avoid using the global scope," and be done with it?

The global scope of a JS program is a rich topic, with much more utility and nuance than you would likely assume. This chapter first explores how the global scope is (still) useful and relevant to writing JS programs today, then looks at differences in where and *how to access* the global scope in different JS environments.

Fully understanding the global scope is critical in your mastery of using lexical scope to structure your programs.

Why Global Scope?

It's likely no surprise to readers that most applications are composed of multiple (sometimes many!) individual JS files. So how exactly do all those separate files get stitched together in a single runtime context by the JS engine?

With respect to browser-executed applications, there are three main ways.

First, if you're directly using ES modules (not transpiling them into some other module-bundle format), these files are loaded individually by the JS environment. Each module then `import` s references to whichever other modules it needs to access. The separate module files cooperate with each other exclusively through these shared imports, without needing any shared outer scope.

Second, if you're using a bundler in your build process, all the files are typically concatenated together before delivery to the browser and JS engine, which then only processes one big file. Even with all the pieces of the application co-located in a single file, some mechanism is necessary for each piece to register a *name* to be referred to by other pieces, as well as some facility for that access to occur.

In some build setups, the entire contents of the file are wrapped in a single enclosing scope, such as a wrapper function, universal module (UMD—see Appendix A), etc. Each piece can register itself for access from other pieces by way of local variables in that shared scope. For example:

```
(function wrappingOuterScope(){
    var moduleOne = (function one(){
        // ..
    })();

    var moduleTwo = (function two(){
        // ..

        function callModuleOne() {
            moduleOne.someMethod();
        }

        // ..
    })();
})();
```

As shown, the `moduleOne` and `moduleTwo` local variables inside the `wrappingOuterScope()` function scope are declared so that these modules can access each other for their cooperation.

While the scope of `wrappingOuterScope()` is a function and not the full environment global scope, it does act as a sort of "application-wide scope," a bucket where all the top-level identifiers can be stored, though not in the real global scope. It's kind of like a stand-in for the global scope in that respect.

And finally, the third way: whether a bundler tool is used for an application, or whether the (non-ES module) files are simply loaded in the browser individually (via `<script>` tags or other dynamic JS resource loading), if there is no single surrounding scope encompassing all these pieces, the **global scope** is the only way for them to cooperate with each other:

A bundled file of this sort often looks something like this:

```
var moduleOne = (function one(){
    // ..
})();
var moduleTwo = (function two(){
    // ..

    function callModuleOne() {
        moduleOne.someMethod();
    }

    // ..
})();
```

Here, since there is no surrounding function scope, these `moduleOne` and `moduleTwo` declarations are simply dropped into the global scope. This is effectively the same as if the files hadn't been concatenated, but loaded separately:

module1.js:

```
var moduleOne = (function one(){
    // ..
})();
```

module2.js:

```
var moduleTwo = (function two(){
    // ..

    function callModuleOne() {
        moduleOne.someMethod();
    }

    // ..
})();
```

If these files are loaded separately as normal standalone .js files in a browser environment, each top-level variable declaration will end up as a global variable, since the global scope is the only shared resource between these two separate files—they're independent programs, from the perspective of the JS engine.

In addition to (potentially) accounting for where an application's code resides during runtime, and how each piece is able to access the other pieces to cooperate, the global scope is also where:

- JS exposes its built-ins:
  - primitives: `undefined`, `null`, `Infinity`, `NaN`
  - natives: `Date()`, `Object()`, `String()`, etc.
  - global functions: `eval()`, `parseInt()`, etc.
  - namespaces: `Math`, `Atomics`, `JSON`
  - friends of JS: `Intl`, `WebAssembly`
- The environment hosting the JS engine exposes its own built-ins:
  - `console` (and its methods)
  - the DOM (`window`, `document`, etc)
  - timers (`setTimeout(..)`, etc)
  - web platform APIs: `navigator`, `history`, geolocation, WebRTC, etc.

These are just some of the many *globals* your programs will interact with.

NOTE:

Node also exposes several elements "globally," but they're technically not in the `global` scope: `require()`, `__dirname`, `module`, `URL`, and so on.

Most developers agree that the global scope shouldn't just be a dumping ground for every variable in your application. That's a mess of bugs just waiting to happen. But it's also undeniable that the global scope is an important *glue* for practically every JS application.

Where Exactly is this Global Scope?

It might seem obvious that the global scope is located in the outermost portion of a file; that is, not inside any function or other block. But it's not quite as simple as that.

Different JS environments handle the scopes of your programs, especially the global scope, differently. It's quite common for JS developers to harbor misconceptions without even realizing it.

Browser "Window"

With respect to treatment of the global scope, the most *pure* environment JS can be run in is as a standalone .js file loaded in a web page environment in a browser. I don't mean "pure" as in nothing automatically added—lots may be added!—but rather in terms of minimal intrusion on the code or interference with its expected global scope behavior.

Consider this .js file:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!
```

This code may be loaded in a web page environment using an inline `<script>` tag, a `<script src=..>` script tag in the markup, or even a dynamically created `<script>` DOM element. In all three cases, the `studentName` and `hello` identifiers are declared in the global scope.

That means if you access the global object (commonly, `window` in the browser), you'll find properties of those same names there:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ window.studentName }!`);
}

window.hello();
// Hello, Kyle!
```

That's the default behavior one would expect from a reading of the JS specification: the outer scope *is* the global scope and `studentName` is legitimately created as global variable.

That's what I mean by *pure*. But unfortunately, that won't always be true of all JS environments you encounter, and that's often surprising to JS developers.

Globals Shadowing Globals

Recall the discussion of shadowing (and global unshadowing) from Chapter 3, where one variable declaration can override and prevent access to a declaration of the same name from an outer scope.

An unusual consequence of the difference between a global variable and a global property of the same name is that, within just the global scope itself, a global object property can be shadowed by a global variable:

```
window.something = 42;

let something = "Kyle";

console.log(something);
// Kyle

console.log(window.something);
// 42
```

The `let` declaration adds a `something` global variable but not a global object property (see Chapter 3). The effect then is that the `something` lexical identifier shadows the `something` global object property.

It's almost certainly a bad idea to create a divergence between the global object and the global scope. Readers of your code will almost certainly be tripped up.

A simple way to avoid this gotcha with global declarations: always use `var` for globals. Reserve `let` and `const` for block scopes (see "Scoping with Blocks" in Chapter 6).

DOM Globals

I asserted that a browser-hosted JS environment has the most *pure* global scope behavior we'll see. However, it's not entirely *pure*.

One surprising behavior in the global scope you may encounter with browser-based JS applications: a DOM element with an `id` attribute automatically creates a global variable that references it.

Consider this markup:

```
<ul id="my-todo-list">
   <li id="first">Write a book</li>
   ..
</ul>
```

And the JS for that page could include:

```
first;
// <li id="first">..</li>

window["my-todo-list"];
// <ul id="my-todo-list">..</ul>
```

If the `id` value is a valid lexical name (like `first`), the lexical variable is created. If not, the only way to access that global is through the global object (`window[..]`).

The auto-registration of all `id`-bearing DOM elements as global variables is an old legacy browser behavior that nevertheless must remain because so many old sites still rely on it. My advice is never to use these global variables, even though they will always be silently created.

What's in a (Window) Name?

Another global scope oddity in browser-based JS:

```
var name = 42;

console.log(name, typeof name);
// "42" string
```

`window.name` is a pre-defined "global" in a browser context; it's a property on the global object, so it seems like a normal global variable (yet it's anything but "normal").

We used `var` for our declaration, which **does not** shadow the pre-defined `name` global property. That means, effectively, the `var` declaration is ignored, since there's already a global scope object property of that name. As we discussed earlier, had we used `let name`, we would have shadowed `window.name` with a separate global `name` variable.

But the truly surprising behavior is that even though we assigned the number `42` to `name` (and thus `window.name`), when we then retrieve its value, it's a string `"42"`! In this case, the weirdness is because `name` is actually a pre-defined getter/setter on the `window` object, which insists on its value being a string value. Yikes!

With the exception of some rare corner cases like DOM element ID's and `window.name`, JS running as a standalone file in a browser page has some of the most *pure* global scope behavior we will encounter.

Web Workers

Web Workers are a web platform extension on top of browser-JS behavior, which allows a JS file to run in a completely separate thread (operating system wise) from the thread that's running the main JS program.

Since these Web Worker programs run on a separate thread, they're restricted in their communications with the main application thread, to avoid/limit race conditions and other complications. Web Worker code does not have access to the DOM, for example. Some web APIs are, however, made available to the worker, such as `navigator`.

Since a Web Worker is treated as a wholly separate program, it does not share the global scope with the main JS program. However, the browser's JS engine is still running the code, so we can expect similar *purity* of its global scope behavior. Since there is no DOM access, the `window` alias for the global scope doesn't exist.

In a Web Worker, the global object reference is typically made using `self`:

```
var studentName = "Kyle";
let studentID = 42;

function hello() {
    console.log(`Hello, ${ self.studentName }!`);
}

self.hello();
// Hello, Kyle!

self.studentID;
// undefined
```

Just as with main JS programs, `var` and `function` declarations create mirrored properties on the global object (aka, `self`), where other declarations (`let`, etc) do not.

So again, the global scope behavior we're seeing here is about as *pure* as it gets for running JS programs; perhaps it's even more *pure* since there's no DOM to muck things up!

Developer Tools Console/REPL

Recall from Chapter 1 in *Get Started* that Developer Tools don't create a completely adherent JS environment. They do process JS code, but they also lean in favor of the UX interaction being most friendly to developers (aka, developer experience, or DX).

In some cases, favoring DX when typing in short JS snippets, over the normal strict steps expected for processing a full JS program, produces observable differences in code behavior between programs and tools. For example, certain error conditions applicable to a JS program may be relaxed and not displayed when the code is entered into a developer tool.

With respect to our discussions here about scope, such observable differences in behavior may include:

- The behavior of the global scope
- Hoisting (see Chapter 5)
- Block-scoping declarators (`let` / `const`, see Chapter 6) when used in the outermost scope

Although it might seem, while using the console/REPL, that statements entered in the outermost scope are being processed in the real global scope, that's not quite accurate. Such tools typically emulate the global scope position to an extent; it's emulation, not strict adherence. These tool environments prioritize developer convenience, which means that at times (such as with our current discussions regarding scope), observed behavior may deviate from the JS specification.

The take-away is that Developer Tools, while optimized to be convenient and useful for a variety of developer activities, are **not** suitable environments to determine or verify explicit and nuanced behaviors of an actual JS program context.

ES Modules (ESM)

ES6 introduced first-class support for the module pattern (covered in Chapter 8). One of the most obvious impacts of using ESM is how it changes the behavior of the observably top-level scope in a file.

Recall this code snippet from earlier (which we'll adjust to ESM format by using the `export` keyword):

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

export hello;
```

If that code is in a file that's loaded as an ES module, it will still run exactly the same. However, the observable effects, from the overall application perspective, will be different.

Despite being declared at the top level of the (module) file, in the outermost obvious scope, `studentName` and `hello` are not global variables. Instead, they are module-wide, or if you prefer, "module-global."

However, in a module there's no implicit "module-wide scope object" for these top-level declarations to be added to as properties, as there is when declarations appear in the top-level of non-module JS files. This is not to say that global variables cannot exist or be accessed in such programs. It's just that global variables don't get *created* by declaring variables in the top-level scope of a module.

The module's top-level scope is descended from the global scope, almost as if the entire contents of the module were wrapped in a function. Thus, all variables that exist in the global scope (whether they're on the global object or not!) are available as lexical identifiers from inside the module's scope.

ESM encourages a minimization of reliance on the global scope, where you import whatever modules you may need for the current module to operate. As such, you less often see usage of the global scope or its global object.

However, as noted earlier, there are still plenty of JS and web globals that you will continue to access from the global scope, whether you realize it or not!

Node

One aspect of Node that often catches JS developers off-guard is that Node treats every single .js file that it loads, including the main one you start the Node process with, as a *module* (ES module or CommonJS module, see Chapter 8). The practical effect is that the top level of your Node programs **is never actually the global scope**, the way it is when loading a non-module file in the browser.

As of time of this writing, Node has recently added support for ES modules. But additionally, Node has from its beginning supported a module format referred to as "CommonJS", which looks like this:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

module.exports.hello = hello;
```

Before processing, Node effectively wraps such code in a function, so that the `var` and `function` declarations are contained in that wrapping function's scope, **not** treated as global variables.

Envision the preceding code as being seen by Node as this (illustrative, not actual):

```
function Module(module,require,__dirname,...) {
    var studentName = "Kyle";

    function hello() {
        console.log(`Hello, ${ studentName }!`);
    }

    hello();
    // Hello, Kyle!

    module.exports.hello = hello;
}
```

Node then essentially invokes the added `Module(..)` function to run your module. You can clearly see here why `studentName` and `hello` identifiers are not global, but rather declared in the module scope.

As noted earlier, Node defines a number of "globals" like `require()`, but they're not actually identifiers in the global scope (nor properties of the global object). They're injected in the scope of every module, essentially a bit like the parameters listed in the `Module(..)` function declaration.

So how do you define actual global variables in Node? The only way to do so is to add properties to another of Node's automatically provided "globals," which is ironically called `global`. `global` is a reference to the real global scope object, somewhat like using `window` in a browser JS environment.

Consider:

```
global.studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

module.exports.hello = hello;
```

Here we add `studentName` as a property on the `global` object, and then in the `console.log(..)` statement we're able to access `studentName` as a normal global variable.

Remember, the identifier `global` is not defined by JS; it's specifically defined by Node.

Global This

Reviewing the JS environments we've looked at so far, a program may or may not:

- Declare a global variable in the top-level scope with `var` or `function` declarations—or `let`, `const`, and `class`.
- Also add global variables declarations as properties of the global scope object if `var` or `function` are used for the declaration.
- Refer to the global scope object (for adding or retrieving global variables, as properties) with `window`, `self`, or `global`.

I think it's fair to say that global scope access and behavior is more complicated than most developers assume, as the preceding sections have illustrated. But the complexity is never more obvious than in trying to nail down a universally applicable reference to the global scope object.

Yet another "trick" for obtaining a reference to the global scope object looks like:

```
const theGlobalScopeObject =
    (new Function("return this"))();
```

NOTE:

A function can be dynamically constructed from code stored in a string value with the `Function()` constructor, similar to `eval(..)` (see "Cheating: Runtime Scope Modifications" in Chapter 1). Such a function will automatically be run in non-strict-mode (for legacy reasons) when invoked with the normal `()` function invocation as shown; its `this` will point at the global object. See the third book in the series, *Objects & Classes*, for more information on determining `this` bindings.

So, we have `window`, `self`, `global`, and this ugly `new Function(..)` trick. That's a lot of different ways to try to get at this global object. Each has its pros and cons.

Why not introduce yet another!?!?

As of ES2020, JS has finally defined a standardized reference to the global scope object, called `globalThis`. So, subject to the recency of the JS engines your code runs in, you can use `globalThis` in place of any of those other approaches.

We could even attempt to define a cross-environment polyfill that's safer across pre-`globalThis` JS environments, such as:

```
const theGlobalScopeObject =
    (typeof globalThis != "undefined") ? globalThis :
    (typeof global != "undefined") ? global :
    (typeof window != "undefined") ? window :
    (typeof self != "undefined") ? self :
    (new Function("return this"))();
```

Phew! That's certainly not ideal, but it works if you find yourself needing a reliable global scope reference.

(The proposed name `globalThis` was fairly controversial while the feature was being added to JS. Specifically, I and many others felt the "this" reference in its name was misleading, since the reason you reference this object is to access to the global scope, never to access some sort of global/default `this` binding. There were many other names considered, but for a variety of reasons ruled out. Unfortunately, the name chosen ended up as a last resort. If you plan to interact with the global scope object in your programs, to reduce confusion, I strongly recommend choosing a better name, such as (the laughably long but accurate!) `theGlobalScopeObject` used here.)

Globally Aware

The global scope is present and relevant in every JS program, even though modern patterns for organizing code into modules de-emphasizes much of the reliance on storing identifiers in that namespace.

Still, as our code proliferates more and more beyond the confines of the browser, it's especially important we have a solid grasp on the differences in how the global scope (and global scope object!) behave across different JS environments.

With the big picture of global scope now sharper in focus, the next chapter again descends into the deeper details of lexical scope, examining how and when variables can be used.