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

Chapter 3: Working With Scope

NOTE: |
:— |
Work in progress |

Through Chapters 1 and 2, we defined *lexical scope* as the set of rules (determined at compile time) for how the identifiers/variables in a program are organized into units of scope (functions, blocks), as well as how lookups of these identifiers works during run-time.

For conceptual understanding, lexical scope was illustrated with several metaphors: marbles & buckets (bubbles!), conversations, and a tall office building.

Now it's time to sift through a bunch of nuts and bolts of working with lexical scope in our programs. There's a lot more to scope than you probably think. This is one of those chapters that really hammers home just how much we all *don't know* about scope.

TIP: |
:— |
This chapter is very long and detailed. Make sure to take your time working through it, and practice the concepts and code frequently. Don't rush! |

Nested Scopes, Revisited

Again, recall our running example program:

```
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// Suzy
```

What color is the `students` variable reference in the `for`-loop?

In Chapter 2, we described the run-time access of a variable as a "lookup", where the *Engine* has to start by asking the current scope's *Scope Manager* if it knows about an identifier/variable, and proceeding upward/outward back through the chain of nested scopes (toward the global scope) until found, if ever. The lookup stops as soon as the first matching named declaration in a scope bucket is found.

The lookup process thus determines that `students` is a RED marble. And `studentID` in the `if`-statement is determined to be a BLUE marble.

"Lookup" Is (Mostly) Conceptual

This description of the run-time lookup process works for conceptual understanding, but it's not generally how things work in practice.

The color of a marble (what bucket it comes from) – the meta information of what scope a variable originates from – is *usually* known during the initial compilation processing. Because of how lexical scope works, a marble's color will not change based on anything that can happen during run-time.

Since the marble's color is known from compilation, and it's immutable, this information will likely be stored with (or at least accessible from) each variable's entry in the AST; that information is then used in generating the executable instructions that constitute the program's run-time. In other words, *Engine* (from Chapter 2) doesn't need to lookup and figure out which scope bucket a variable comes from. That information is already known!

Avoiding the need for a run-time lookup is a key optimization benefit for lexical scope. Scope is fixed at author-time/compile-time, and unaffected by run-time conditions, so no run-time lookup is necessary. Run-time is operates more performantly without spending time on these lookups.

But I said "…usually known…" just now with respect to a marble's color determination during compilation. In what case would it *not* be known during compilation?

Consider a reference to a variable that isn't declared in any lexically available scopes in the current file – see *Get Started*, Chapter 1, which asserts that each file is its own separate program from the perspective of JS compilation. If no declaration is found, that's not *necessarily* an error. Another file (program) in the run-time may indeed declare that variable in the shared global scope. So the ultimate determination of whether the variable was ever appropriately declared in some available bucket may need to be deferred to the run-time.

The take-away? Any reference to a variable in our program that's initially *undeclared* is left as an uncolored marble during that file's compilation; this color cannot be determined until other relevant file(s) have been compiled and the application run-time begins.

In that respect, some sort of run-time "lookup" for the variable would need to resolve the color of this uncolored marble. If the variable was eventually discovered in the global scope bucket, the color of the global scope thus applies. But this run-time deferred lookup would only be needed once at most, since nothing else during run-time could later change that marble's color.

NOTE: |
:— |
Chapter 2 "Lookup Failures" covers what happens if a marble remains uncolored as its reference is executed. |

Shadowing

Our running example for these chapters uses different variable names across the scope boundaries. Since they all have unique names, in a way it wouldn't matter if all of them were just in one bucket (like RED).

Where having different lexical scope buckets starts to matter more is when you have two or more variables, each in different scopes, with the same lexical names. In such a case, it's very relevant how the different scope buckets are laid out.

Consider:

```
var studentName = "Suzy";

function printStudent(studentName) {
    studentName = studentName.toUpperCase();
    console.log(studentName);
}

printStudent("Frank");
// FRANK

printStudent(studentName);
// SUZY

console.log(studentName);
// Suzy
```

TIP: |
:— |
Before you move on, take some time to analyze this code using the various techniques/metaphors we've covered in the book. In particular, make sure to identify the marble/bubble colors in this snippet. It's good practice! |

The `studentName` variable on line 1 (the `var studentName = ..` statement) creates a RED marble. The same named variable is declared as a BLUE marble on line 3, the parameter in the `printStudent(..)` function definition.

So the question is, what color marble is being referenced in the `studentName = studentName.toUpperCase()` statement, and indeed the next statement, `console.log(studentName)`? All 3 `studentName` references here will be BLUE. Why?

With the conceptual notion of the "lookup", we asserted that it starts with the current scope and works its way outward/upward, stopping as soon as a matching variable is found. The BLUE `studentName` is found right away. The RED `studentName` is never even considered.

This is a key component of lexical scope behavior, called *shadowing*. The BLUE `studentName` variable (parameter) shadows the RED `studentName`. So, the parameter shadows (or is shadowing) the shadowed global variable. Repeat that sentence to yourself a few times to make sure you have the terminology straight!

That's why the re-assignment of `studentName` affects only the inner (parameter) variable, the BLUE `studentName`, not the global RED `studentName`.

When you choose to shadow a variable from an outer scope, one direct impact is that from that scope inward/downward (through any nested scopes), it's now impossible for any marble to be colored as the shadowed variable (RED, in this case). In other words, any `studentName` identifier reference will mean that parameter variable, never the global `studentName` variable. It's lexically impossible to reference the global `studentName` anywhere inside of the `printStudent(..)` function (or any inner scopes it may contain).

Global Unshadowing Trick It is still possible to access a global variable, but not through a typical lexical identifier reference.

In the global scope (RED), `var` declarations and `function`-declarations also expose themselves as properties (of the same name as the identifier) on the *global object* – essentially an object representation of the global scope. If you've done JS coding in a browser environment, you probably identify the global object as `window`. That's not *entirely* accurate, but it's good enough for us to use in discussion for now. In a bit, we'll explore the global scope/object topic more.

Consider this program, specifically executed as a standalone .js file in a browser environment:

```
var studentName = "Suzy";

function printStudent(studentName) {
    console.log(studentName);
    console.log(window.studentName);
}

printStudent("Frank");
// "Frank"
// "Suzy"
```

Notice the `window.studentName` reference? This expression is accessing the global variable `studentName` as a property on `window` (which we're pretending for now is synonomous with the global object). That's the only way to access a shadowed variable from inside the scope where the shadowing variable is present.

WARNING: |
:— |
Leveraging this technique is not very good practice, as it's limited in utility, confusing for readers of your code, and likely to invite bugs to your program. Don't shadow a global variable that you need to access, and conversely, don't access a global variable that you've shadowed. |

The `window.studentName` is a mirror of the global `studentName` variable, not a snapshot copy. Changes to one are reflected in the other, in either direction. Think of `window.studentName` as a getter/setter that accesses the actual `studentName` variable. As a matter of fact, you can even *add* a variable to the global scope by creating/setting a property on the global object (`window`).

This little "trick" only works for accessing a global scope variable (that was declared with `var` or `function`). Other forms of global scope variable declarations do not create mirrored global object properties:

```
var one = 1;
let notOne = 2;
const notTwo = 3;
class notThree {}

console.log(window.one);       // 1
console.log(window.notOne);    // undefined
console.log(window.notTwo);    // undefined
console.log(window.notThree);  // undefined
```

Variables (no matter how they're declared!) that exist in any other scope than the global scope are completely inaccessible from an inner scope where they've been shadowed.

```
var special = 42;

function lookingFor(special) {
    // `special` in this scope is inaccessible from
    // inside keepLooking()

    function keepLooking() {
        var special = 3.141592;
        console.log(special);
        console.log(window.special);
    }

    keepLooking();
}

lookingFor(112358132134);
// 3.141592
// 42
```

The global RED `special` is shadowed by the BLUE `special` (parameter), and the BLUE `special` is itself shadowed by the GREEN `special` inside `keepLooking()`. We can still access RED `special` indirectly as `window.special`.

Copying Is Not Accessing I've been asked the following "But what about…?" question dozens of times, so I'm just going to address it before you even ask!

```
var special = 42;

function lookingFor(special) {
    var another = {
        special: special
    };

    function keepLooking() {
        var special = 3.141592;
        console.log(special);
        console.log(another.special);  // Ooo, tricky!
        console.log(window.special);
    }

    keepLooking();
}
```

```
lookingFor(112358132134);
// 3.141592
// 112358132134
// 42
```

Oh! So does this `another` technique prove me wrong in my above claim of the `special` parameter being "completely inaccessible" from inside `keepLooking()`? No, it does not.

`special: special` is copying the value of the `special` parameter variable into another container (a property of the same name). Of course if you put a value in another container, shadowing no longer applies (unless `another` was shadowed, too!). But that doesn't mean we're accessing the parameter `special`, it means we're accessing the value it had at that moment, but by way of another container (object property). We cannot, for example, reassign that BLUE `special` to another value from inside `keepLooking()`.

Another "But…!?" you may be about to raise: what if I'd used objects or arrays as the values instead of the numbers (`112358132134`, etc)? Would us having references to objects instead of copies of primitive values "fix" the inaccessibility? No. Mutating the contents of the object value via such a reference copy is **not** the same thing as lexically accessing the variable itself. We still couldn't reassign the BLUE `special`.

Illegal Shadowing Not all combinations of declaration shadowing are allowed. One case to be aware of is that `let` can shadow `var`, but `var` cannot shadow `let`.

Consider:

```
function something() {
    var special = "JavaScript";
    {
        let special = 42;   // totally fine shadowing
        // ..
    }
}

function another() {
    // ..
    {
        let special = "JavaScript";
        {
            var special = "JavaScript";   // Syntax Error
            // ..
        }
    }
}
```

Notice in the `another()` function, the inner `var special` declaration is attempting to declare a function-wide `special`, which in and of itself is fine (as shown by the `something()` function).

The Syntax Error description in this case indicates that `special` has already been defined, but that error message is a little misleading (again, no such error happens in `something()`, as shadowing is generally allowed just fine). The real reason it's raised as a Syntax Error is because the `var` is basically trying to "cross the boundary" of the `let` declaration of the same name, which is not allowed.

The boundary crossing effectively stops at each function boundary, so this variant raises no exception:

```
function another() {
    // ..
    {
        let special = "JavaScript";

        whatever(function callback(){
            var special = "JavaScript";   // totally fine shadowing
            // ..
        });
    }
}
```

Just remember: `let` can shadow `var`, but not the other way around.

Function Name Scope

As you're probably aware, a `function` declaration looks like this:

```
function askQuestion() {
    // ..
}
```

And as discussed in Chapter 1 and 2, such a `function` declaration will create a variable in the enclosing scope (in this case, the global scope) named `askQuestion`.

What about this program?

```
var askQuestion = function() {
    // ..
};
```

The same thing is true with respect to the variable `askQuestion` being created. But since we have a `function` expression – a function definition used as value instead of as a declaration – this function definition will not “hoist” (covered later in this chapter).

But hoisting is only one difference between `function` declarations and `function` expressions. The other major difference is what happens to the name identifier on the function.

Consider the assignment of a named `function` expression:

```
var askQuestion = function ofTheTeacher(){
    // ..
};
```

We know `askQuestion` ends up in the outer scope. But what about the `ofTheTeacher` identifier? For `function` declarations, the name identifier ends up in the outer/enclosing scope, so it would seem reasonable to assume that’s the case here. But it’s not.

`ofTheTeacher` is declared as a variable **inside the function itself**:

```
var askQuestion = function ofTheTeacher() {
    console.log(ofTheTeacher);
};

askQuestion();
// function ofTheTeacher()...

console.log(ofTheTeacher);
// ReferenceError: 'ofTheTeacher' is not defined
```

Not only is `ofTheTeacher` declared inside the function rather than outside, but it’s also created as read-only:

```
var askQuestion = function ofTheTeacher() {
    "use strict";
    ofTheTeacher = 42;   // this assignment fails

    //..
};

askQuestion();
// TypeError
```

Because we used strict mode, the assignment failure is reported as a Type Error; in non-strict mode, such an assignment fails silently with no exception.

What about when a `function` expression has no name identifier?

```
var askQuestion = function(){
   // ..
};
```

A `function` expression with a name identifier is referred to as a “named function expression”, and one without a name identifier is referred to as an “anonymous function expression”. Anonymous function expressions have no name identifier, and so have no effect on either the outer/enclosing scope or their own.

NOTE: |
:— |
We’ll discuss named vs. anonymous `function` expressions in much more detail, including what factors affect the decision to use one or the other, in Appendix A. |

Arrow Functions

ES6 added an additional `function` expression form, called “arrow functions”:

```
var askQuestion = () => {
    // ..
};
```

The `=>` arrow function doesn’t require the word `function` to define it. Also, the `( .. )` around the parameter list is optional in some simple cases. Likewise, the `{ .. }` around the function body is optional in some simple cases. And when the `{ .. }` are omitted, a return value is computed without using a `return` keyword.

NOTE: |
:— |
The attractiveness of `=>` arrow functions is often sold as “shorter syntax”, and that’s claimed to equate to objectively more readable functions. This claim is dubious at best, and outright misguided in general. We’ll dig into the “readability” of function forms in Appendix A. |

Arrow functions are lexically anonymous, meaning they have no directly related identifier that references the function. The assignment to `askQuestion` creates an inferred name of “askQuestion”, but that’s **not the same thing as being non-anonymous**:

```
var askQuestion = () => {
    // ..
};

askQuestion.name;   // askQuestion
```

Arrow functions achieve their syntactic brevity at the expense of having to mentally juggle a bunch of variations for different forms/conditions. Just a few for example:

```
() => 42

id => id.toUpperCase()

(id,name) => ({ id, name })

(...args) => {
    return args[args.length - 1];
};
```

The real reason I bring up arrow functions is because of the common but incorrect claim that arrow functions somehow behave differently with respect to lexical scope from standard `function` functions.

This is incorrect.

Other than being anonymous (and having no declarative form), arrow functions have the same rules with respect to lexical scope as `function` functions do. An arrow function, with or without `{ .. }` around its body, still creates a separate, inner nested bucket of scope. Variable declarations inside this nested scope bucket behave the same as in `function` functions.

Why Global Scope?

We've referenced the "global scope" a number of times already, but we should dig into that topic in more detail. We'll start by exploring whether the global scope is (still) useful and relevant to writing JS programs, and then look at differences in how the global scope is *found* in different JS environments.

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 run-time context by the JS engine?

With respect to browser-executed applications, there are 3 main ways:

1. If you're exclusively using ES modules (not transpiling those into some other module-bundle format), then 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 scopes.

2. 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 being 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 be made.

   In some approaches, the entire contents of the file are wrapped in a single enclosing scope (such as a wrapper function, UMD-like module, etc), so each piece can register itself for access by other pieces by way of local variables in that shared scope.

   For example:

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

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

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

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

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

   While the scope of `outerScope()` 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, even if not in the real global scope. So it's kind of like a stand-in for the global scope in that respect.

3. Whether a bundler 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 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 processed in the global scope. This is effectively the same as if the file hadn't been concatenated:

module1.js:

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

module2.js:

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

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

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

Again, if these files are loaded 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 (programs, from the perspective of the JS engine).

In addition to (potentially) accounting for where an application's code resides during run-time, 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 that is hosting JS exposes its built-ins:
    - `console` (and its methods)
    - the DOM (`window`, `document`, etc)
    - timers (`setTimeout(..)`, etc)
    - web platform APIs: `navigator`, `history`, geolocation, WebRTC, etc

  NOTE: |
  :— |
  Node also exposes several elements "globally", but they're technically not in its `global` scope: `require()`, `__dirname`, `module`, `URL`, etc. |

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 virtually 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, in particular the global scope, differently. It's extremely common for JS developers to have misconceptions in this regard.

Browser "Window"

With resepct to treatment of the global scope, the most *pure* (not completely!) 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 behavior.

Consider this simple .js file:

```
var studentName = "Kyle";

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

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

This code may be loaded in a webpage 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. That's what I mean by *pure*. That won't always be true of other JS environments, and that's often surprising to JS developers.

Shadowing Revisited Recall the discussion of shadowing from earlier? An unusual consequence of the difference between a global variable and a global property of the same name is that a global object property can be shadowed by a global variable:

```
window.something = 42;

let something = "Kyle";

console.log(something);
// Kyle
```

The `let` declaration adds a `something` global variable, which shadows the `something` global object property.

While it's *possible* to shadow in this manner, it's almost certainly a bad idea to do so. Don't create a divergence between the global object and the global scope.

What's In A Name? I asserted that this browser-hosted JS environment has the most *pure* global scope behavior we'll see. Things are not entirely *pure*, however.

Consider:

```
var name = 42;

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

`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 (though it's anything but "normal"). We used `var` for the declaration, which doesn't 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 in the previous section, had we use `let name`, we would have shadowed `window.name` with a separate global `name` variable.

But the truly weird behavior is that even though we assigned the number `42` to `name`, when we then retrieve its value, it's a string `"42"`! In this case, the weirdness is because `window.name` is actually a getter/setter on the global object, which insists on a string value. Wow!

With the exception some rare corner cases like `window.name`, JS running as a standalone file in a browser page has some of the most *pure* global scope behavior we're likely to encounter.

Web Workers

Web Workers are a web platform extension for typical 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 browser-hosted JS.

Since these web worker programs run on a separate thread, they're restricted in their communications with the main application thread, to avoid/control 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. But there is no DOM access, so the `window` alias for the global scope doesn't exist.

In a web worker, a global object reference is typically made with `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.

Developer Tools Console/REPL

Recall from "Get Started" Chapter 1 that Developer Tools don't create a completely authentic JS environment. They do process JS code, but they also bend the UX of the interaction in favor of being friendly to developers (aka, "Developer Experience", DX).

In many cases, favoring DX when entering short JS snippets over the normal strict steps expected for processing a full JS program produces observable differences in behavior of code. 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 (discussed later in this chapter), and block-scoping declarators (`let` / `const`, see Chapter 4) when used in the outermost scope.

Even though while using the console/REPL it seems like statements entered in the outermost scope are being processed in the real global scope, that's not strictly accurate. The tool emulates that to an extent, but 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 being very convenient and useful for a variety of developer activities, are **not** suitable environments to determine or verify some of the explicit and nuanced behaviors of an actual JS program context.

ES Modules (ESM)

ES6 introduced first-class support for the module pattern (which we'll cover more in Chapter 6). One of the most obvious impacts of using ESM is how it changes the behavior of observably top-level scope in a file.

Recall this code snippet from earlier:

```
var studentName = "Kyle";

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

hello();
// Hello, Kyle!

export hello;
```

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

Despite being declared at the top-level of the (module) file, the outermost obvious scope, `studentName` and `hello` are not global variables. Instead, they are module-wide, or if you prefer, "module-global". They are not added to any global scope object, nor are they added to any accessible "module-global" object.

This is not to say that global variables cannot exist 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

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

```
var studentName = "Kyle";

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

hello();
// Hello, Kyle!

module.exports.hello = hello;
```

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

Think of the above code when processed by Node sorta like 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 (again, essentially) invokes the `Module(..)` function to run your module. You can clearly see here why `studentName` and `hello` identifiers are thus 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. They’re provided in the available scope to every module, essentially a bit like the parameters listed to this `Module(..)` function above.

WARNING: |
:— |
The part 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*, so this wrapping always occurs! That means that your main Node program file does **not** act (with respect to scope) like a .js file otherwise loaded as the main program in a browser environment! |

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 called `global`. `global` is ostensibly (if not actually) a reference to the real global scope object.

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, `global` is not defined by JS, it’s defined by Node.

Global This

Reviewing where we’ve been so far, depending on which JS environment our code is running in, a program may or may not be able to:

- 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` were 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 articulate a broadly applicable reference to the global scope object.

Another "trick" for getting a reliable reference to this global scope object might look like:

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

NOTE: |
:— |
A function that is dynamically constructed with the `Function()` constructor will automatically be run in non-strict mode (for legacy reasons) when invoked as shown (the normal `()` function invocation); thus, its `this` will be the global object. See Book 3 *Objects & Classes* for more information. |

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

Why not introduce yet another!?!?

At the time of this writing, JS recently introduced a standardized reference to the global scope object, called `globalThis`. So, depending on the recency of the JS engines your code runs in, you can then use `globalThis` in place of any of those other approaches.

You might even attempt a cross-environment approach that's safer across older JS environments pre-`globalThis`, something like:

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

Phew! At least now you're more aware of the breadth of topic on the global scope and global scope object.

When Can I Use A Variable?

At what point does a variable become available to use in a certain part of a program? There may seem to be an obvious answer: *after* the variable has been declared/created. Right? Not quite.

Consider:

```
greeting();
// Hello!

function greeting() {
    console.log("Hello!");
}
```

This code works fine. You may have seen or even written code like it before. But did you ever wonder how or why it works? Specifically, why can you access the identifier `greeting` from line 1 (to retrieve and execute a function reference), even though the `greeting()` function declaration doesn't occur until line 3?

Recall how Chapter 1 pointed out that all identifiers are registered to their respective scopes during compile time. Moreover, every identifier is *created* at the beginning of the scope it belongs to, **every time that scope is entered**.

The term for registering a variable at the top of its enclosing scope, even though its declaration may appear further down in the scope, is called **hoisting**.

But hoisting alone doesn't fully answer the posed question. Sure, we can see an identifier called `greeting` from the beginning of the scope, but why can we **call** the `greeting()` function before it's been declared?

In other words, how does `greeting` have any value in it, like the function reference, when the scope first begins? That's an additional characteristic of `function` declarations, called "function hoisting". When a `function` declaration's name identifier is registered at the top of a scope, it is additionally initialized to that function's reference.

Function hoisting only applies to formal `function` declarations (which appear outside of blocks – see FiB in Chapter 4), not to `function` expression assignments. Consider:

```
greeting();
// Type Error

var greeting = function greeting() {
    console.log("Hello!");
};
```

Line one (`greeting();`) throws an error. But the *kind* of error thrown is very important to notice. A Type Error means we're trying to do something with a value that is not allowed. Indeed, the error message would, depending on your JS environment, say something like " 'undefined' is not a function", or alternately, " 'greeting' is not a function".

We should notice that the error is **not** a Reference Error. It's not telling us that it couldn't find `greeting` as an identifier in the scope. It's telling us that `greeting` doesn't hold a function reference at that moment.

What does it hold?

Variables declared with `var` are, in addition to being hoisted, also automatically initialized to `undefined` at the beginning of the scope. Once they're initialized, they're available to be used (assigned to, retrieved, etc). So on that first line, `greeting` exists, but it holds only the default `undefined` value. It's not until line 3 that `greeting` gets assigned the function reference.

Pay close attention to the distinction here. A `function` declaration is hoisted and initialized to its function value (again, called "function hoisting"). By contrast, a `var` variable is hoisted, but it's only auto-initialized to `undefined`. Any subsequent `function` expression assignments to that variable don't happen until that statement is reached during run-time execution.

In both cases, the name of the identifier is hoisted. But the value association doesn't get handled at initialization time unless the identifier came from a `function` declaration.

Let's look at another example of "variable hoisting":

```
greeting = "Hello!";
console.log(greeting);
// Hello!

var greeting = "Howdy!";
```

The `greeting` variable is available to be assigned to by the time we reach line 1. Why? There's two necessary parts: the identifier was hoisted, and it was automatically initialized to `undefined`.

NOTE: |
:— |
Variable hoisting probably feels a bit unnatural to use in a program. But is function hoisting also a bad idea? We'll explore this in more detail in Appendix A. |

Yet Another Metaphor

Chapter 2 was full of metaphors (to illustrate scope), but here we are faced with yet another: hoisting itself is a metaphor. It's a visualization of how JS handles variable and `function` declarations.

When most people explain what "hoisting" means, they will describe "lifting" – like lifting a heavy weight upward – the identifiers all the way to the top of a scope. Typically, they will assert that the JS engine will *rewrite* that program before it executes it, so that it looks more like this:

```
var greeting;           // hoisted declaration moved to the top

greeting = "Hello!";    // the original line 1
console.log(greeting);
// Hello!

greeting = "Howdy!";    // ‘var‘ is gone!
```

The hoisting metaphor proposes that JS pre-processes the original program and re-arranges it slightly, so that all the declarations have been moved to the top of their respective scopes, before execution. Moreover, the hoisting metaphor asserts that `function` declarations are, in their entirety, hoisted to the top of each scope, as well.

Consider:

```
studentName = "Suzy"
greeting();
// Hello Suzy!

function greeting() {
    console.log(‘Hello ${ studentName }!‘);
}

var studentName;
```

The "rule" of the hoisting metaphor is that function declarations get hoisted first, then variables immediately after all the functions. Thus, hoisting suggests that program is *re-written* by the JS engine to look like this:

```
function greeting() {
    console.log(‘Hello ${ studentName }!‘);
}
var studentName;

studentName = "Suzy";
greeting();
// Hello Suzy!
```

The hoisting metaphor is convenient. Its benefit is allowing us to hand wave over the magical look-ahead pre-processing necessary to find all these declarations buried deep in scopes and somehow move (hoist) them to the top; we can then think about the program as if it's executed by the JS engine in a **single pass**, top-down. Single-pass seems more straightforward than Chapter 1's assertion of a 2-phase processing.

Hoisting as re-ordering code may be an attractive simplification, but it's not accurate. The JS engine doesn't actually rewrite the code. It can't magically look-ahead and find declarations. The only way to accurately find them, as well as all the scope boundaries in the program, would be to fully parse the code. Guess what parsing is? The first phase of the 2-phase processing! There's no magical mental gymnastics that gets around that fact.

So if "hoisting" as a metaphor is inaccurate, what should we do with the term? It's still useful – indeed, even members of TC39 regularly use it! – but we shouldn't think of it as the re-ordering of code.

WARNING: |
:— |
Incorrect or incomplete mental models may seem sufficient because they can occasionally lead to accidental right answers. But in the long run it's harder to accurately analyze and predict outcomes if you're not thinking closely to how the JS engine works. |

"Hoisting" should refer to the **compile-time operation** of generating run-time instructions for the automatic registration of a variable at the beginning of its scope, each time that scope is entered.

Re-declaration?

What do you think happens when variable is declared more than once in the same scope?

Consider:

```
var studentName = "Frank";

console.log(studentName);
// Frank

var studentName;

console.log(studentName);
// ???
```

What do you expect to be printed as that second message? Many think the second `var studentName` has re-declared the variable (and "reset" it), so they expect `undefined` to be printed.

But is there such a thing as a variable being "re-declared" in the same scope? No.

If you consider this program from the perspective of the hoisting metaphor, the code would be re-ordered like this for execution purposes:

```
var studentName;
var studentName;    // this is clearly a pointless no-op!

studentName = "Frank";
console.log(studentName);
// Frank

console.log(studentName);
// Frank
```

Since hoisting is actually about registering a variable at the beginning of a scope, there's nothing to be done in the middle of the scope where the original program actually had the second `var studentName` statement. It's just a no-op(eration), a dead pointless statement.

TIP: |
:— |
In our conversation-style from Chapter 2, *Compiler* would find the second `var` declaration statement and ask the *Scope Manager* if it had already seen a `studentName` identifier; since it had, there wouldn't be anything else to do. |

It's also important to point out that `var studentName;` doesn't mean `var studentName = undefined;`, as most people assume. Let's prove they're different by considering this variation of the program:

```
var studentName = "Frank";

console.log(studentName);
// Frank

var studentName = undefined;   // let's add the initialization explicitly

console.log(studentName);
// undefined
```

See how the explicit `= undefined` initialization produces a different outcome than assuming it still happens implicitly even if omitted? In the next section, we'll revisit this topic of initialization of variables from their declarations.

So a repeated `var` declaration of the same identifier name in a scope is effectively a do-nothing statement. What about repeating a declaration within a scope using `let` or `const`?

```
let studentName = "Frank";

console.log(studentName);

let studentName = "Suzy";
```

This program will not execute, but instead immediately throw a Syntax Error. Depending on your JS environment, the error message will indicate something like: "Identifier 'studentName' has already been declared." In other words, this is a case where attempted "re-declaration" is explicitly not allowed!

It's not just that two declarations involving `let` will throw this error. If either declaration uses `let`, the other can be either `let` or `var`, and an error will still occur, as illustrated with these two variations:

```
var studentName = "Frank";
let studentName = "Suzy";
```

```
let studentName = "Frank";
var studentName = "Suzy";
```

In both cases, a Syntax Error is thrown on the *second* declaration. In other words, the only way to "re-declare" a variable is to use `var` for all (two or more) of its declarations.

But why disallow it? The reason for the error is not technical per se, as `var` "re-declaration" has always been allowed; clearly, the same allowance could have been made for `let`. But it's really more of a "social engineering" issue. "Re-declaration" of variables is seen by some, including many on the TC39 body, as a bad habit that can lead to program bugs.

So when ES6 introduced `let`, they decided to prevent "re-declaration" with an error. When *Compiler* asks *Scope Manager* about a declaration, if that identifier has already been declared, and if either/both declarations were made with `let`, an error is thrown. The intended signal to the developer is, "Stop relying on sloppy re-declaration!".

NOTE: |
:— |
This is of course a stylistic opinion, not really a technical argument. Many developers agree with it, and that's probably in part why TC39 included the error (as well as conforming to `const`). But a reasonable case could have been made that staying consistent with `var`'s precedent was more prudent, and that such opinion-enforcement was best left to opt-in tooling like linters. We'll explore whether `var` (and its associated behavior) can still be useful in Appendix A. |

Constants? The `const` keyword is a little more constrained than `let`. Like `let`, `const` cannot be repeated with the same identifier in the same scope. But there's actually an overriding technical reason why that sort of "re-declaration" is disallowed, unlike `let` which disallows "re-declaration" mostly for stylistic reasons.

The `const` keyword requires a variable to be initialized:

```
const empty;   // SyntaxError
```

`const` declarations create variables that cannot be re-assigned:

```
const studentName = "Frank";
console.log(studentName);
// Frank

studentName = "Suzy";   // TypeError
```

The `studentName` variable cannot be re-assigned because it's declared with a `const`.

WARNING: |
:— |
The error thrown when re-assigning to `studentName` is a Type Error, not a Syntax Error. The subtle distinction here is actually pretty important, but unfortunately far too easy to miss. Syntax Errors represent faults in the program that stop it from even starting execution. Type Errors represent faults that arise during program execution. In the above snippet, `"Frank"` is printed out before we process the re-assignment of `studentName`, which then throws the error. |

So if `const` declarations cannot be re-assigned, and `const` declarations always require assignments, then we have a clear technical reason why `const` must disallow any "re-declarations":

```
const studentName = "Frank";
const studentName = "Suzy";   // obviously this must be an error
```

Since `const` "re-declaration" must be disallowed (on technical grounds), TC39 essentially felt that `let` "re-declaration" should be disallowed as well.

Loops So it's clear from our previous discussion that JS doesn't really want us to "re-declare" our variables within the same scope. That probably seems like a straightforward admonition, until you consider what it means repeated execution of declaration statements in loops.

Consider:

```
var keepGoing = true;

while (keepGoing) {
    let value = Math.random();
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

Is `value` being "re-declared" repeatedly in this program? Will we get errors thrown?

No.

All the rules of scope (including "re-declaration" of `let`-created variables) are applied *per scope instance*. In other words, each time a scope is entered during execution, everything resets.

Each loop iteration is its own new scope instance, and within each scope instance, `value` is only being declared once. So there's no attempted "re-declaration", and thus no error.

Before we consider other loop forms, what if the `value` declaration in the previous snippet were changed to a `var`?

```
var keepGoing = true;

while (keepGoing) {
    var value = Math.random();
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

Is `value` being "re-declared" here, especially since we know `var` allows it? No. Because `var` is not treated as a block-scoping declaration (see Chapter 4), it attaches itself to the global scope. So there's just one `value`, in the same (global, in this case) scope as `keepGoing`. No "re-declaration"!

One way to keep this all straight is to remember that `var`, `let`, and `const` do not exist in the code by the time it starts to execute. They're handled entirely by the compiler.

What about "re-declaration" with other loop forms, like `for`-loops?

```
for (let i = 0; i < 3; i++) {
    let value = i * 10;
    console.log(`${ i }: ${ value }`);
}
// 0: 0
// 1: 10
// 2: 20
```

It should be clear that there's only one `value` declared per scope instance. But what about `i`? Is it being "re-declared"?

To answer that, consider what scope `i` is in? It might seem like it would be in the outer (in this case, global) scope, but it's not. It's in the scope of `for`-loop body, just like `value` is. In fact, you could sorta think about that loop in this more verbose equivalent form:

```
{
    let $$i = 0;  // a fictional variable for illustration

    for ( ; $$i < 3; $$i++) {
        let i = $$i;   // here's our actual loop ‘i‘!
        let value = i * 10;
        console.log(‘${ i }: ${ value }‘);
    }
    // 0: 0
    // 1: 10
    // 2: 20
}
```

Now it should be clear: the illustrative `$$i`, as well as `i` and `value` variables, are all declared exactly once per scope instance. No "re-declaration" here.

What about other `for`-loop forms?

```
for (let index in students) {
    // this is fine
}

for (let student of students) {
    // so is this
}
```

Same thing with `for..in` and `for..of` loops: the declared variable is treated as *inside* the loop body, and thus is handled per iteration (aka, per scope instance). No "re-declaration".

OK, I know you're thinking that I sound like a broken record at this point. But let's explore how `const` impacts these looping constructs.

Consider:

```
var keepGoing = true;

while (keepGoing) {
    const value = Math.random();   // ooo, a shiny constant!
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

Just like the `let` variant of this program we saw earlier, `const` is being run exactly once within each loop iteration, so it's safe from "re-declaration" troubles. But things get more complicated when we talk about `for`-loops.

`for..in` and `for..of` are fine to use with `const`:

```
for (const index in students) {
    // this is fine
}

for (const student of students) {
    // this is also fine
}
```

But not the general `for`-loop:

```
for (const i = 0; i < 3; i++) {
    // oops, this is going to fail
    // after the first iteration
}
```

What's wrong here? We could use `let` just fine in this construct, and we asserted that it creates a new `i` for each loop iteration scope, so it doesn't even seem to be a "re-declaration".

Let's mentally "expand" that loop like we did earlier:

```
{
    const $$i = 0;  // a fictional variable for illustration

    for ( ; $$i < 3; $$i++) {
        const i = $$i;   // here's our actual loop ‘i‘!
        // ..
    }
}
```

Do you spot the problem? Our `i` is indeed just created once inside the loop. That's not the problem. The problem is the conceptual `$$i` that must be incremented each time with the `$$i++` expression. That's re-assignment, which isn't allowed for constants.

Remember, this "expanded" form is only a conceptual model to help you intuit the source of the problem. You might wonder if JS could have made the `const $$i = 0` instead into `let $ii = 0`, which would then allow `const` to work with our classic `for`-loop? It's possible, but then it would have been creating potentially surprising exceptions to `for`-loop semantics.

In other words, it's a rather arbitrary (and likely confusing) nuanced exception to allow `i++` in the `for`-loop header to skirt strictness the `const` assignment, but not allow other re-assignments of `i` inside the loop iteration, as is sometimes done. As such, the more straightforward answer is: `const` can't be used with the classic `for`-loop form because of the re-assignment.

Interestingly, if you don't do re-assignment, then it's valid:

```
var keepGoing = true;

for (const i = 0; keepGoing; ) {
    keepGoing = (Math.random() > 0.5);
    // ..
}
```

This is silly. There's no reason to declare `i` in that position with a `const`, since the whole point of such a variable in that position is **to be used for counting iterations**. Just use a different loop form, like a `while` loop.

Uninitialized

With `var` declarations, the variable is "hoisted" to the top of its scope. But it's also automatically initialized to the `undefined` value, so that the variable can be used throughout the entire scope.

However, `let` and `const` declarations are not quite the same in this respect.

Consider:

```
console.log(studentName);
// ReferenceError

let studentName = "Suzy";
```

The result of this program is that a Reference Error is thrown on the first line. Depending on your JS environment, the error message may say something like: "Cannot access 'studentName' before initialization."

NOTE: |
:— |
The error message as seen here used to be much more vague or misleading. Thankfully, I and others were successfully able to lobby for JS engines to improve this error message so it tells you what's wrong. |

That error message is very instructive as to what's wrong. `studentName` exists on line 1, but it's not been initialized, so it cannot be used yet. Let's try this:

```
studentName = "Suzy";   // let's try to initialize it!
// ReferenceError

console.log(studentName);

let studentName;
```

Oops. We still get the Reference Error, but now on the first line where we're trying to assign to (aka, initialize!) this so-called "uninitialized" variable `studentName`. What's the deal!?

The real question is, how do we initialize an uninitialized variable? For `let` / `const`, the **only way** to do so is with the assignment attached to a declaration statement. An assignment by itself is insufficient!

Consider:

```
// some other code

let studentName = "Suzy";

console.log(studentName);
// Suzy
```

Here, we are initializing the `studentName`, in this case to `"Suzy"` instead of `undefined`, by way of the `let` declaration statement form that's coupled with an assignment.

Alternately:

```
// ..

let studentName;
// or:
// let studentName = undefined;

// ..
```

```
studentName = "Suzy";

console.log(studentName);
// Suzy
```

NOTE: |
:— |
That's interesting! Recall from earlier, we said that `var studentName;` is *not* the same as `var studentName = undefined;`, but here with `let`, they behave the same. The difference comes down to the fact that `var studentName` automatically initializes at the top of the scope, where `let studentName` does not. |

Recall that we asserted a few times so far that *Compiler* ends up removing any `var` / `let` / `const` declaration statements, replacing them with the instructions at the top of each scope to register the appropriate identifiers.

So if we analyze what's going on here, we see that an additional nuance is that *Compiler* is also adding an instruction in the middle of the program, at the point where the variable `studentName` was declared, to do the auto-initialization. We cannot use the variable at any point prior to that initialization occuring. The same goes for `const` as it does for `let`.

The term coined by TC39 to refer to this *period of time* from the entering of a scope to where the auto-initialization of the variable occurs, is: Temporal Dead Zone (TDZ). The TDZ is the time window where a variable exists but is still uninitialized, and therefore cannot be accessed in any way. Only the execution of the instructions left by *Compiler* at the point of the original declaration can do that initialization. After that moment, the TDZ is over, and the variable is free to be used for the rest of the scope.

By the way, "temporal" in TDZ does indeed refer to *time* not *position-in-code*. Consider:

```
askQuestion();
// ReferenceError

let studentName = "Suzy";

function askQuestion() {
    console.log(`${ studentName }, what do you think?`);
}
```

Even though positionally the `console.log(..)` referencing `studentName` comes *after* the `let studentName` declaration, timing wise the `askQuestion()` function is invoked *before*, while `studentName` is still in its TDZ!

Many have claimed that TDZ means `let` and `const` do not hoist. But I think this is an inaccurate, or at least misleading, claim. I think the real difference with `let` and `const` is that they do not get automatically initialized, the way `var` does. The *debate* then is if the auto-initialization is *part of* hoisting, or not? I think auto-registration of a variable at the top of the scope (i.e., what I call "hoisting") and auto-initialization are separate and shouldn't be lumped together under the term single term "hoisting".

We already know `let` and `const` don't auto-intialize at the top of the scope. But let's prove that `let` and `const` *do* hoist (auto-register at the top of the scope), courtesy of our friend shadowing (see earlier in this chapter):

```
var studentName = "Kyle";

{
    console.log(studentName);
    // ???

    // ..

    let studentName = "Suzy";

    console.log(studentName);
    // Suzy
}
```

What's going to happen with the first `console.log(..)` statement? If `let studentName` didn't hoist to the top of the scope, then it *should* print `"Kyle"`, right? At that moment, it seems, only the outer `studentName` would exist.

But instead, we're going to get a TDZ error at that first `console.log(..)`, because in fact, the inner scope's `studentName` **was** hoisted (auto-registered at the top of the scope). But what **didn't** happen (yet!) was the auto-initialization of that inner `studentName`; it's still unintialized at that moment, hence the TDZ violation!

So to summarize, TDZ errors occur because `let` / `const` declarations *do* hoist their declarations to the top of their scopes, but unlike `var`, they defer the auto-initialization of their variables until the moment in the code's sequencing where the original declaration appeared. This window of time, whatever its length, is the TDZ.

How can you avoid TDZ errors? My advice: always put your `let` and `const` declarations at the top of any scope. Shrink the TDZ window to zero (or near zero) time, and then it'll be moot.

Why is TDZ even a thing? Why didn't TC39 dictate that `let` / `const` auto-initialize the way `var` does? We'll cover the *why* of TDZ in Appendix A.

Scope Closed

Phew! That was quite a long and involved chapter! Take some deep breaths and try to let that all sink in. OK, now take a few more.

Before moving on, let me just remind you once again: this stuff is complex and challenging. It's OK if you're feeling mentally worn out – I certainly am after writing it. Don't rush to keep reading, but instead take your time to review the material from this chapter, analyze your own code, and practice these techniques.

The importance specifically of block scope deserves its own full discussion, so that's where we turn our attention next.