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

Chapter 5: The (Not So) Secret Lifecycle of Variables

By now you should have a decent grasp of the nesting of scopes, from the global scope downward—called a program's scope chain.

But just knowing which scope a variable comes from is only part of the story. If a variable declaration appears past the first statement of a scope, how will any references to that identifier *before* the declaration behave? What happens if you try to declare the same variable twice in a scope?

JS's particular flavor of lexical scope is rich with nuance in how and when variables come into existence and become available to the program.

When Can I Use a Variable?

At what point does a variable become available to use within its scope? 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 4?

Recall Chapter 1 points 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 most commonly used for a variable being visible from the beginning 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 question. 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 the variable `greeting` have any value (the function reference) assigned to it, from the moment the scope starts running? The answer is a special characteristic of formal `function` declarations, called *function hoisting*. When a `function` declaration's name identifier is registered at the top of its scope, it's additionally auto-initialized to that function's reference. That's why the function can be called throughout the entire scope!

One key detail is that both *function hoisting* and `var`-flavored *variable hoisting* attach their name identifiers to the nearest enclosing **function scope** (or, if none, the global scope), not a block scope.

NOTE:

Declarations with `let` and `const` still hoist (see the TDZ discussion later in this chapter). But these two declaration forms attach to their enclosing block rather than just an enclosing function as with `var` and `function` declarations. See "Scoping with Blocks" in Chapter 6 for more information.

Hoisting: Declaration vs. Expression

Function hoisting only applies to formal `function` declarations (specifically those which appear outside of blocks—see "FiB" in Chapter 6), not to `function` expression assignments. Consider:

```
greeting();
// TypeError

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

Line 1 (`greeting();`) throws an error. But the *kind* of error thrown is very important to notice. A `TypeError` means we're trying to do something with a value that is not allowed. Depending on your JS environment, the error message would say something like, "'undefined' is not a function," or more helpfully, "'greeting' is not a function."

Notice that the error is **not** a `ReferenceError`. JS isn't telling us that it couldn't find `greeting` as an identifier in the scope. It's telling us that `greeting` was found but doesn't hold a function reference at that moment. Only functions can be invoked, so attempting to invoke some non-function value results in an error.

But what does `greeting` hold, if not the function reference?

In addition to being hoisted, variables declared with `var` are also automatically initialized to `undefined` at the beginning of their scope—again, the nearest enclosing function, or the global. Once initialized, they're available to be used (assigned to, retrieved from, etc.) throughout the whole scope.

So on that first line, `greeting` exists, but it holds only the default `undefined` value. It's not until line 4 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*). A `var` variable is also hoisted, and then auto-initialized to `undefined`. Any subsequent `function` expression assignments to that variable don't happen until that assignment is processed during runtime execution.

In both cases, the name of the identifier is hoisted. But the function reference association isn't handled at initialization time (beginning of the scope) unless the identifier was created in a formal `function` declaration.

Variable Hoisting

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

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

var greeting = "Howdy!";
```

Though `greeting` isn't declared until line 5, it's available to be assigned to as early as line 1. Why?

There's two necessary parts to the explanation:

- the identifier is hoisted,
- **and** it's automatically initialized to the value `undefined` from the top of the scope.

> NOTE:
>
> Using *variable hoisting* of this sort probably feels unnatural, and many readers might rightly want to avoid relying on it in their programs. But should all hoisting (including *function hoisting*) be avoided? We'll explore these different perspectives on hoisting in more detail in Appendix A.

Hoisting: Yet Another Metaphor

Chapter 2 was full of metaphors (to illustrate scope), but here we are faced with yet another: hoisting itself. Rather than hoisting being a concrete execution step the JS engine performs, it's more useful to think of hoisting as a visualization of various actions JS takes in setting up the program **before execution**.

The typical assertion of what hoisting means: *lifting*—like lifting a heavy weight upward—any identifiers all the way to the top of a scope. The explanation often asserted is that the JS engine will actually *rewrite* that program before execution, so that it looks more like this:

```
var greeting;           // hoisted declaration
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 a bit, 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. Consider:

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

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

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

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

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

This 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 just think about the program as if it's executed by the JS engine in a **single pass**, top-down.

Single-pass definitely seems more straightforward than Chapter 1's assertion of a two-phase processing.

Hoisting as a mechanism for re-ordering code may be an attractive simplification, but it's not accurate. The JS engine doesn't actually re-arrange 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 two-phase processing! There's no magical mental gymnastics that gets around that fact.

So if the hoisting metaphor is (at best) inaccurate, what should we do with the term? I think it's still useful—indeed, even members of TC39 regularly use it!—but I don't think we should claim it's an actual re-arrangement of source code.

WARNING:

Incorrect or incomplete mental models often still 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 your thinking isn't particularly aligned with how the JS engine works.

I assert that hoisting *should* be used to refer to the **compile-time operation** of generating runtime instructions for the automatic registration of a variable at the beginning of its scope, each time that scope is entered.

That's a subtle but important shift, from hoisting as a runtime behavior to its proper place among compile-time tasks.

Re-declaration?

What do you think happens when a 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 for that second message? Many believe the second `var studentName` has re-declared the variable (and thus "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-arranged like this for execution purposes:

```
var studentName;
var studentName;    // 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 pointless statement.

TIP:

In the style of the conversation narrative 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 assume. Let's prove they're different by considering this variation of the program:

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

var studentName;
console.log(studentName);   // Frank <--- still!

// let's add the initialization explicitly
var studentName = undefined;
console.log(studentName);   // undefined <--- see!?
```

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

A repeated `var` declaration of the same identifier name in a scope is effectively a do-nothing operation. Here's another illustration, this time across a function of the same name:

```
var greeting;

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

// basically, a no-op
var greeting;

typeof greeting;        // "function"

var greeting = "Hello!";

typeof greeting;        // "string"
```

The first `greeting` declaration registers the identifier to the scope, and because it's a `var` the auto-initialization will be `undefined`. The `function` declaration doesn't need to re-register the identifier, but because of *function hoisting* it overrides the auto-initialization to use the function reference. The second `var greeting` by itself doesn't do anything since `greeting` is already an identifier and *function hoisting* already took precedence for the auto-initialization.

Actually assigning `"Hello!"` to `greeting` changes its value from the initial function `greeting()` to the string; `var` itself doesn't have any effect.

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 `SyntaxError`. Depending on your JS environment, the error message will indicate something like: "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 the error will still occur, as illustrated with these two variations:

```
var studentName = "Frank";

let studentName = "Suzy";
```

and:

```
let studentName = "Frank";

var studentName = "Suzy";
```

In both cases, a `SyntaxError` 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`.

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.

NOTE:

This is of course a stylistic opinion, not really a technical argument. Many developers agree with the position, and that's probably in part why TC39 included the error (as well as `let` 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. In Appendix A, we'll explore whether `var` (and its associated behavior, like "re-declaration") can still be useful in modern JS.

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!"

Constants?

The `const` keyword is 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, so omitting an assignment from the declaration results in a `SyntaxError`:

```
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 `studentName` is a `TypeError`, not a `SyntaxError`. 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 preceding 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": any `const` "re-declaration" would also necessarily be a `const` re-assignment, which can't be allowed!

```
const studentName = "Frank";

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

Since `const` "re-declaration" must be disallowed (on those technical grounds), TC39 essentially felt that `let` "re-declaration" should be disallowed as well, for consistency. It's debatable if this was the best choice, but at least we have the reasoning behind the decision.

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 for 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 6), it attaches itself to the global scope. So there's just one `value` variable, in the same scope as `keepGoing` (global scope, in this case). No "re-declaration" here, either!

One way to keep this all straight is to remember that `var`, `let`, and `const` keywords are effectively *removed* from the code by the time it starts to execute. They're handled entirely by the compiler.

If you mentally erase the declarator keywords and then try to process the code, it should help you decide if and when (re-)declarations might occur.

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:

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

    for ( /* nothing */; $$i < 3; $$i++) {
        // here's our actual loop `i`!
        let i = $$i;

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

Now it should be clear: the `i` and `value` variables are both 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:

```js
var keepGoing = true;
while (keepGoing) {
    // ooo, a shiny constant!
    const value = Math.random();
    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`:

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

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

But not the general `for`-loop:

```js
for (const i = 0; i < 3; i++) {
    // oops, this is going to fail with
    // a Type Error 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:

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

    for ( ; $$i < 3; $$i++) {
        // here's our actual loop `i`!
        const i = $$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** (not "re-declaration"), 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 effectively 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 could have introduced potentially surprising exceptions to `for`-loop semantics.

For example, it would have been a rather arbitrary (and likely confusing) nuanced exception to allow `i++` in the `for`-loop header to skirt strictness of the `const` assignment, but not allow other re-assignments of `i` inside the loop iteration, as is sometimes useful.

The straightforward answer is: `const` can't be used with the classic `for`-loop form because of the required re-assignment.

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

```js
var keepGoing = true;

for (const i = 0; keepGoing; /* nothing here */ ) {
    keepGoing = (Math.random() > 0.5);
    // ..
}
```

That works, but it's pointless. 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, or use a `let`!

Uninitialized Variables (aka, TDZ)

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 `ReferenceError` 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, several of us in the community were successfully able to lobby for JS engines to improve this error message so it more accurately tells you what's wrong!

That error message is quite indicative of 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 `ReferenceError`, 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 an assignment attached to a declaration statement. An assignment by itself is insufficient! Consider:

```
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.

Alternatively:

```
// ..

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.

Remember that we've asserted a few times so far that *Compiler* ends up removing any `var` / `let` / `const` declarators, 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 handle that declaration's 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 done, and the variable is free to be used for the rest of the scope.

A `var` also has technically has a TDZ, but it's zero in length and thus unobservable to our programs! Only `let` and `const` have an observable TDZ.

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 }, do you know?`);
}
```

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

There's a common misconception that TDZ means `let` and `const` do not hoist. This is an inaccurate, or at least slightly misleading, claim. They definitely hoist.

The actual difference is that `let` / `const` declarations do not automatically initialize at the beginning of the scope, 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 at the top of the scope (to `undefined`) are distinct operations and shouldn't be lumped together under the single term "hoisting."

We've already seen that `let` and `const` don't auto-initialize 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 "Shadowing" in Chapter 3):

```
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 the first `console.log(..)` *should* print `"Kyle"`, right? At that moment, it would seem, only the outer `studentName` exists, so that's the variable `console.log(..)` should access and print.

But instead, the first `console.log(..)` throws a TDZ error, because in fact, the inner scope's `studentName` **was** hoisted (auto-registered at the top of the scope). What **didn't** happen (yet!) was the auto-initialization of that inner `studentName`; it's still uninitialized 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 (hint: temporal), 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) length, and then it'll be moot.

But why is TDZ even a thing? Why didn't TC39 dictate that `let` / `const` auto-initialize the way `var` does? Just be patient, we'll come back to explore the *why* of TDZ in Appendix A.

Finally Initialized

Working with variables has much more nuance than it seems at first glance. *Hoisting*, *(re)declaration*, and the *TDZ* are common sources of confusion for developers, especially those who have worked in other languages before coming to JS. Before moving on, make sure your mental model is fully grounded on these aspects of JS scope and variables.

Hoisting is generally cited as an explicit mechanism of the JS engine, but it's really more a metaphor to describe the various ways JS handles variable declarations during compilation. But even as a metaphor, hoisting offers useful structure for thinking about the life-cycle of a variable—when it's created, when it's available to use, when it goes away.

Declaration and re-declaration of variables tend to cause confusion when thought of as runtime operations. But if you shift to compile-time thinking for these operations, the quirks and *shadows* diminish.

The TDZ (temporal dead zone) error is strange and frustrating when encountered. Fortunately, TDZ is relatively straightforward to avoid if you're always careful to place `let` / `const` declarations at the top of any scope.

As you successfully navigate these twists and turns of variable scope, the next chapter will lay out the factors that guide our decisions to place our declarations in various scopes, especially nested blocks.