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

Chapter 1: How is Scope Determined?

NOTE: |
:— |
Work in progress |

One key foundation of programming languages is storing values in variables and retrieving those values later. In fact, this process is the primary way we model program *state*. A program without *state* is a pretty uninteresting program, so this topic is at the very heart of what it means to write programs.

But to understand how variables work, we should first ask: where do variables *live*? In other words, how are your program's variables organized and accessed by the JS engine?

These questions imply the need for a well-defined set of rules for managing variables, called *scope*. The first step to understanding scope is discuss how the JS engine processes and executes a program.

About This Book

If you already finished *Get Started* (the first book of this series), you're in the right spot! If not, before you proceed I encourage you to *start there* for the best foundation.

Our focus in this second book is the first pillar (of three!) in the JS language: the scope system and its function closures, and how these mechanisms enable the module design pattern.

JS is typically classified as an interpreted scripting language, so it's assumed that programs are processed in a single, top-down pass. But JS is in fact parsed/compiled in a separate phase **before execution begins**. The code author's decisions on where to place variables, functions, and blocks with respect to each other are analyzed according to the rules of scope, during the initial parsing/compilation phase. The resulting scope is unaffected by run-time conditions.

JS functions are themselves values, meaning they can be assigned and passed around just like numbers or strings. But since these functions hold and access variables, they maintain their original scope no matter where in the program the functions are eventually executed. This is called closure.

Modules are a pattern for code organization characterized by a collection of public methods that have access (via closure) to variables and functions that are hidden inside the internal scope of the module.

Compiling Code

Depending on your level of experience with various languages, you may be surprised to learn that JS is compiled, even though it's typically labeled as "dynamic" or "interpreted". JS is *not* typically compiled well in advance, as are many traditionally-compiled languages, nor are the results of compilation portable among various distributed JS engines. Nevertheless, the JS engine essentially performs similar steps as other traditional language compilers.

The reason we need to talk about code compilation to understand scope is because JS's scope is entirely determined during this phase.

A program is generally processed by a compiler in three basic stages:

1. **Tokenizing/Lexing:** breaking up a string of characters into meaningful (to the language) chunks, called tokens. For instance, consider the program: `var a = 2;`. This program would likely be broken up into the following tokens: `var`, `a`, `=`, `2`, and `;`. Whitespace may or may not be persisted as a token, depending on whether it's meaningful or not.

NOTE: |
:— |
The difference between tokenizing and lexing is subtle and academic, but it centers on whether or not these tokens are identified in a *stateless* or *stateful* way. Put simply, if the tokenizer were to invoke stateful parsing rules to figure out whether `a` should be considered a distinct token or just part of another token, *that* would be **lexing**. |

2. **Parsing:** taking a stream (array) of tokens and turning it into a tree of nested elements, which collectively represent the grammatical structure of the program. This tree is called an "AST" (Abstract Syntax Tree).

   For example, the tree for `var a = 2;` might start with a top-level node called `VariableDeclaration`, with a child node called `Identifier` (whose value is `a`), and another child called `AssignmentExpression` which itself has a child called `NumericLiteral` (whose value is `2`).

3. **Code Generation:** taking an AST and turning it into executable code. This part varies greatly depending on the language, the platform it's targeting, etc.

   The JS engine takes our above described AST for `var a = 2;` and turns it into a set of machine instructions to actually *create* a variable called `a` (including reserving memory, etc.), and then store a value into `a`.

NOTE: |
:— |
The implementation details of a JS engine (utilizing system memory resources, etc) is much deeper than we will dig here. We'll keep our focus on the observable behavior of our programs and let the JS engine manage those system abstractions.
|

The JS engine is vastly more complex than *just* those three stages. In the process of parsing and code-generation, there are steps to optimize the performance of the execution, including collapsing redundant elements, etc. In fact, code can even be re-compiled and re-optimized during the progression of execution.

So, I'm painting only with broad strokes here. But you'll see shortly why *these* details we *do* cover, even at a high level, are relevant.

JS engines don't have the luxury of plenty of time to optimize, because JS compilation doesn't happen in a build step ahead of time, as with other languages. It usually must happen in mere microseconds (or less!) right before the code is executed. To ensure the fastest performance under these constraints, JS engines use all kinds of tricks (like JITs, which lazy compile and even hot re-compile, etc.) which are well beyond the "scope" of our discussion here.

Required: Two Phases

To state it as simply as possible, a JS program is processed in (at least) two phases: parsing/compilation first, then execution.

The breakdown of a parsing/compilation phase separate from the subsequent execution phase is observable fact, not theory or opinion. While the JS specification does not require "compilation" explicitly, it requires behavior which is essentially only practical in a compile-then-execute cadence.

There are three program characteristics you can use to prove this to yourself: syntax errors, "early errors", and hoisting (covered in Chapter 3).

Consider this program:

```
var greeting = "Hello";
console.log(greeting);
greeting = ."Hi";
// SyntaxError: unexpected token .
```

This program produces no output (`"Hello"` is not printed), but instead throws a `SyntaxError` about the unexpected `.` token right before the `"Hi"` string. Since the syntax error happens after the well-formed `console.log(..)` statement, if JS was executing top-down line by line, one would expect the `"Hello"` message being printed before the syntax error being thrown. That doesn't happen. In fact, the only way the JS engine could know about the syntax error on the third line, before executing the first and second lines, is because the JS engine first parses this entire program before any of it is executed.

Next, consider:

```
console.log("Howdy");
saySomething("Hello","Hi");
// Uncaught SyntaxError: Duplicate parameter name not allowed in this context

function saySomething(greeting,greeting) {
    "use strict";
    console.log(greeting);
}
```

The "Howdy" message is not printed, despite being a well-formed statement. Instead, just like the previous snippet, the SyntaxError here is thrown before the program is executed. In this case, it's because strict-mode (opted in for only the saySomething(..) function in this program) forbids, among many other things, functions to have duplicate parameter names; this has always been allowed in non-strict mode. This is not a syntax error in the sense of being a malformed string of tokens (like ."Hi" above), but is required by the specification to be thrown as an "early error" for strict-mode programs.

How does the JS engine know that the greeting parameter has been duplicated? How does it know that the saySomething(..) function is even in strict-mode while processing the parameter list (the "use strict" pragma appears only in the function body)? Again, the only reasonable answer to these questions is that the code must first be parsed before execution.

Finally, consider:

```
function saySomething() {
    var greeting = "Hello";
    {
        greeting = "Howdy";
        let greeting = "Hi";
        console.log(greeting);
    }
}

saySomething();
// ReferenceError: Cannot access 'greeting' before initialization
```

The noted ReferenceError occurs on the line with the statement greeting = "Howdy". What's being indicated is that the greeting variable for that statement is the one from the next line, let greeting = "Hi", rather than from the previous statement var greeting = "Hello".

The only way the JS engine could know, at the line where the error is thrown, that the *next statement* would declare a block-scoped variable of the same name (`greeting`) – which creates the conflict of accessing the variable too early, while in its so called "TDZ", Temporal Dead Zone (see Chapter 3) – is if the JS engine had already processed this code in an earlier pass, and already set up all the scopes and their variable associations. This processing of scopes and declarations can only accurately be done by parsing the program before execution, and it's called "hoisting" (see Chapter 3).

WARNING: |
:— |
It's often asserted that `let` and `const` declarations are not hoisted, as an explanation of the occurence of the "TDZ" (Chapter 3) behavior just illustrated. This is not accurate. If these kinds of declarations were not hoisted, then `greeting = "Howdy"` assignment would simply be targetting the `var greeting` variable from the outer (function) scope, with no need to throw an error; the block-scoped `greeting` wouldn't *exist* yet. But the TDZ error itself proves that the block-scoped `greeting` must have been hoisted to the top of that block scope! |

Hopefully you're now convinced that JS programs are parsed before any execution begins. But does that prove they are compiled?

This is an interesting question to ponder. Could JS parse a program, but then execute that program by *interpreting* the AST node-by-node **without** compiling the program in between? Yes, that is *possible*, but it's extremely unlikely, because it would be highly inefficient performance wise. It's hard to imagine a scenario where a production-quality JS engine would go to all the trouble of parsing a program into an AST, but not then convert (aka, "compile") that AST into the most efficient (binary) representation for the engine to then execute.

Many have endeavored to split hairs with this terminology, as there's plenty of nuance to fuel "well, actually…" interjections. But in spirit and in practice, what the JS engine is doing in processing JS programs is **much more alike compilation** than different.

Classifying JS as a compiled language is not about a distribution model for its binary (or byte-code) executable representations, but about keeping a clear distinction in our minds about the phase where JS code is processed and analyzed, which indisputedly happens observably *before* the code starts to be executed. We need proper mental models for how the JS engine treats our code if we want to understand JS effectively.

Compiler Speak

Let's define a simple JS program to analyze over the next few chapters:

```
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
```

Other than declarations, all occurrences of variables/identifiers in a program serve in one of two "roles": either they're the *target* of an assignment or they're the *source* of a value.

NOTE: |
:— |
When I first learned compiler theory in my Computer Science degree, we were taught the terms "LHS" (aka, *target*) and "RHS" (aka, *source*) for these roles. As you might guess from the "L" and the "R", the acronyms mean "Left-Hand Side" and "Right-Hand Side", respectively, as in left and right sides of an `=` assignment operator. However, assignment targets and sources don't always literally appear on the left or right of an `=`, so it's probably less confusing to think in terms of *target* / *source* instead of *left* / *right*. |

How do you know if a variable is a *target*? Check if there is a value anywhere that is being assigned to it; if so, it's a *target*. If not, then the variable is a *source* instead.

Targets

Let's look at our program example with respect to these roles. Consider:

```
students = [
    /* .. */
];
```

This statement is clearly an assignment operation; remember, the `var students` part is handled entirely as a declaration at compile time, and is thus irrelevant during execution. Same with the `nextStudent = getStudentName(73)` statement.

There are three other *target* assignment operations in the code that are perhaps less obvious.

```
for (let student of students) {
```

That statement assigns a value to `student` for each iteration of the loop. Another *target* reference:

```
getStudentName(73)
```

But how is that an assignment to a *target*? Look closely: the argument `73` is assigned to the parameter `studentID`.

And there's one last (subtle) *target* reference in our program. Can you spot it?

..

..

..

Did you identify this one?

```
function getStudentName(studentID) {
```

A `function` declaration is a special case of a *target* reference. You could think of it like `var getStudentName = function(studentID)`, but that's not exactly accurate. An identifier `getStudentName` is declared (at compile-time), but the `= function(studentID)` part is also handled at compilation; the association between `getStudentName` and the function is automatically set up at the beginning of the scope rather than waiting for an `=` assignment statement to be executed.

NOTE: |
:— |
This immediate automatic function assignment from `function` declarations is referred to as "function hoisting", and will be covered in Chapter 3. |

Sources

So we've identified all five *target* references in the program. The other variable references must then be *source* references (because that's the only other option!).

In `for (let student of students)`, we said that `student` is a *target*, but `students` is the *source* reference. In the statement `if (student.id == studentID)`, both `student` and `studentID` are *source* references. `student` is also a *source* reference in `return student.name`.

In `getStudentName(73)`, `getStudentName` is a *source* reference (which we hope resolves to a function reference value). In `console.log(nextStudent)`, `console` is a *source* reference, as is `nextStudent`.

NOTE: |
:— |
In case you were wondering, `id`, `name`, and `log` are all properties, not variable references. |

What's the importance of understanding *targets* vs. *sources*? In Chapter 2, we'll revisit this topic and cover how a variable's role impacts its lookup (specifically, if the lookup fails).

Cheating: Run-Time Scope Modifications

It should be clear by now that scope is determined as the program is compiled, and should not be affected by any run-time conditions. However, in non-strict mode, there are technically still two ways to cheat this rule, and modify the scopes during the run-time.

Neither of these techniques *should* be used – they're both very bad ideas, and you should be using strict mode anyway – but it's important to be aware of them in case you run across code that does.

The `eval(..)` function receives a string of code to compile and execute on the fly during the program run-time. If that string of code has a `var` or `function` declaration in it, those declarations will modify the scope that the `eval(..)` is currently executing in:

```
function badIdea() {
    eval("var oops = 'Ugh!';");
    console.log(oops);
}

badIdea();
// Ugh!
```

If the `eval(..)` had not been present, the `oops` variable in `console.log(oops)` would not exist, and would throw a Reference Error. But `eval(..)` modifies the scope of the `badIdea()` function at run-time. This is a bad idea for many reasons, including the performance hit of modifying the already compiled and optimized scope, every time `badIdea()` runs. Don't do it!

The second cheat is the `with` keyword, which essentially dynamically turns an object into a local scope – its properties are treated as identifiers in that scope's block:

```
var badIdea = {
    oops: "Ugh!"
};

with (badIdea) {
    console.log(oops);
    // Ugh!
}
```

The global scope was not modified here, but `badIdea` was turned into a scope at run-time rather than compile-time. Again, this is a terrible idea, for performance and readability reasons. Don't!

At all costs, avoid `eval(..)` (at least, `eval(..)` creating declarations) and `with`. As mentioned, neither of these cheats is available in strict mode, so if you just use strict mode – you should! – then the temptation is removed.

Lexical Scope

For a language whose scope is determined at compile time, its scope model is called "lexical scope". This term "lexical" is related to the "lexing" stage of compilation. The key concept is that the lexical scope of a program is controlled entirely by the placement of functions, blocks, and scopes, in relation to each other.

If you place a variable declaration inside a function, the compiler handles this declaration as it's parsing the function, and associates that declaration with the function's scope. If a variable is block-scope declared (`let` / `const`), then it's associated with the nearest enclosing `{ .. }` block, rather than its enclosing function (as with `var`).

A reference (*target* or *source*) for a variable must be resolved to coming from one of the scopes that are *lexically available*, otherwise the variable is said to be "undeclared" (which usually results in an error!). If the variable is not in the current scope, the next outer/enclosing scope will be consulted. This process of stepping out one level of scope nesting continues until either a matching variable declaration can be found, or the global scope is reached and there's nowhere else to go.

It's important to note that compilation doesn't actually *do anything* in terms of reserving memory for scopes and variables.

Instead, compilation creates a map of all the lexical scopes that the program will need as it executes. You can think of this plan/map as inserted code that will define all the scopes (aka, “lexical environments”) and register all the identifiers for each scope.

So scopes are planned out during compilation – that’s why we refer to “lexical scope” as a compile-time decision – but they aren’t actually created until run-time. Each scope is instantiated in memory each time it needs to run.

In the next chapter, we’ll build a deeper conceptual understanding of lexical scope.