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

NOTE:
Work in progress

Table of Contents

- Appendix B: Practice

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

Appendix A: Exploring Further

NOTE: |
:— |
Work in progress |

// TODO

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

Appendix B: Practice

// TODO

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.

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

Chapter 2: Understanding Lexical Scope

In Chapter 1, we explored how scope is determined at code compilation, a model called "lexical scope".

Before we get to the nuts and bolts of how using lexical scope in our programs, we should make sure we have a good conceptual foundation for how scope works. This chapter will illustrate *scope* with several metaphors. The goal here is to *think* about how your program is handled by the JS engine in ways that more closely match how the JS engine actually works.

Buckets, and Bubbles, and Marbles... Oh My!

One metaphor I've found effective in understanding scope is sorting colored marbles into buckets of their matching color.

Imagine you come across a pile of marbles, and notice that all the marbles are colored red, blue, or green. To sort all the marbles, let's drop the red ones into a red bucket, green into a green bucket, and blue into a blue bucket. After sorting, when you later need a green marble, you already know the green bucket is where to go to get it.

In this metaphor, the marbles are the variables in our program. The buckets are scopes (functions and blocks), which we just conceptually assign individual colors for our discussion purposes. The color of each marble is thus determined by which *color* scope we find the marble originally created in.

Let's annotate the program example from Chapter 1 with scope color labels:

```
// outer/global scope: RED

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

function getStudentName(studentID) {
    // function scope: BLUE

    for (let student of students) {
```

```
        // loop scope: GREEN

        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

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

We've designated 3 scope colors with code comments: RED (outermost global scope), BLUE (scope of function `getStudentName(..)`), and GREEN (scope of/inside the `for` loop). But it still may be difficult to recognize the boundaries of these scope buckets when looking at a code listing.

Figure 1 tries to make the scope boundaries easier to visualize by drawing colored bubbles around each scope:

Fig. 1: Nested Scope Bubbles

1. **Bubble 1** (RED) encompasses the global scope, which has three identifiers/variables: `students` (line 1), `getStudentName` (line 8), and `nextStudent` (line 16).
2. **Bubble 2** (BLUE) encompasses the scope of the function `getStudentName(..)` (line 8), which has just one identifier/variable: the parameter `studentID` (line 8).
3. **Bubble 3** (GREEN) encompasses the scope of the `for`-loop (line 9), which has just one identifier/variable: `student` (line 9).

Scope bubbles are determined during compilation based on where the functions / blocks of scope are written, the nesting inside each other, etc. Each scope bubble is entirely contained within its parent scope bubble – a scope is never partially in two different outer scopes.

Each marble (variable/identifier) is colored based on which bubble (bucket) it's declared in, not the color of the scope it may be accessed from (e.g., `students` on line 9 and `studentID` on line 10).

NOTE: |
:— |
Remember we asserted in Chapter 1 that `id`, `name`, and `log` are all properties, not variables; in other words, they're not marbles in buckets, so they don't get

colored based on any the rules we're discussing in this book. To understand how such property accesses are handled, see Book 3 *Objects & Classes*. |

As the JS engine processes a program (during compilation), and finds a declaration for a variable, it essentially asks, "which *color* scope (bubble, bucket) am I currently in?" The variable is designated as that same *color*, meaning it belongs to that bucket/bubble.

The GREEN bucket is wholly nested inside of the BLUE bucket, and similarly the BLUE bucket is wholly nested inside the RED bucket. Scopes can nest inside each other as shown, to any depth of nesting as your program needs.

References (non-declarations) to variables/identifiers can be made from either the current scope, or any scope above/outside the current scope, but never to lower/nested scopes. So an expression in the RED bucket only has access to RED marbles, not BLUE or GREEN. An expression in the BLUE bucket can reference either BLUE or RED marbles, not GREEN. And an expression in the GREEN bucket has access to RED, BLUE, and GREEN marbles.

We can conceptualize the process of determining these non-declaration marble colors during runtime as a lookup. Since the `students` variable reference in the `for`-loop statement on line 9 is not a declaration, it has no color. So we ask the current scope bucket (BLUE) if it has a marble matching that name. Since it doesn't, the lookup continues with the next outer/containing scope (RED). The RED bucket has a marble of the name `students`, so the loop-statement's `students` variable is determined to be a RED marble.

The `if (student.id == studentID)` on line 10 is similarly determined to reference a GREEN marble named `student` and a BLUE marble `studentID`.

NOTE: |
:— |
The JS engine doesn't generally determine these marble colors during run-time; the "lookup" here is a rhetorical device to help you understand the concepts. During compilation, most or all variable references will be from already-known scope buckets, so their color is determined at that, and stored with each marble reference to avoid unnecessary lookups as the program runs. More on this in the next chapter. |

The key take-aways from marbles & buckets (and bubbles!):

- Variables are declared in certain scopes, which can be thought of as colored marbles in matching-color buckets.
- Any reference to a variable of that same name in that scope, or any deeper nested scope, will be a marble of that same color – unless an intervening scope "shadows" the variable declaration; see Chapter 3 "Shadowing.
- The determination of colored buckets, and the marbles they contain, happens during compilation. This information is used for variable (marble color) "lookups" during code execution.

A Conversation Among Friends

Another useful metaphor for the process of analyzing variables and the scopes they come from is to imagine various conversations that go on inside the engine as code is processed and then executed. We can "listen in" on these conversations to get a better conceptual foundation for how scopes work.

Let's now meet the members of the JS engine that will have conversations as they process that program:

1. *Engine*: responsible for start-to-finish compilation and execution of our JavaScript program.
2. *Compiler*: one of *Engine*'s friends; handles all the dirty work of parsing and code-generation (see previous section).
3. *Scope Manager*: another friend of *Engine*; collects and maintains a look-up list of all the declared variables/identifiers, and enforces a set of rules as to how these are accessible to currently executing code.

For you to *fully understand* how JavaScript works, you need to begin to *think* like *Engine* (and friends) think, ask the questions they ask, and answer their questions likewise.

To explore these conversations, recall again our running program example:

```
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
```

Let's examine how JS is going to process that program, specifically starting with the first statement. The array and its contents are just basic JS value literals (and thus unaffected by any scoping concerns), so our focus here will be on the `var students = [ .. ]` declaration and initialization-assignment parts.

We typically think of that as a single statement, but that's not how our friend *Engine* sees it. In fact, *Engine* sees two distinct operations, one which *Compiler* will handle during compilation, and the other which *Engine* will handle during execution.

The first thing *Compiler* will do with this program is perform lexing to break it down into tokens, which it will then parse into a tree (AST).

Once *Compiler* gets to code-generation, there's more detail to consider than may be obvious. A reasonable assumption would be that *Compiler* will produce code for the first statement such as: "Allocate memory for a variable, label it `students`, then stick a reference to the array into that variable." But there's more to it.

Here's how *Compiler* will handle that statement:

1. Encountering `var students`, *Compiler* will ask *Scope Manager* to see if a variable named `students` already exists for that particular scope bucket. If so, *Compiler* would ignore this declaration and move on. Otherwise, *Compiler* will produce code that (at execution time) asks *Scope Manager* to create a new variable called `students` in that scope bucket.
2. *Compiler* then produces code for *Engine* to later execute, to handle the `students = []` assignment. The code *Engine* runs will first ask *Scope Manager* if there is a variable called `students` accessible in the current scope bucket. If not, *Engine* keeps looking elsewhere (see "Nested Scope" below). Once *Engine* finds a variable, it assigns the reference of the `[ .. ]` array to it.

In conversational form, the first-phase of compilation for the program might play out between *Compiler* and *Scope Manager* like this:

> ***Compiler***: Hey *Scope Manager* (of the global scope), I found a formal declaration for an identifier called `students`, ever heard of it?
>
> ***(Global) Scope Manager***: Nope, haven't heard of it, so I've just now created it for you.
>
> ***Compiler***: Hey *Scope Manager*, I found a formal declaration for an identifier called `getStudentName`, ever heard of it?
>
> ***(Global) Scope Manager***: Nope, but I just created it for you.

> ***Compiler***: Hey *Scope Manager*, `getStudentName` points to a function, so we need a new scope bucket.
>
> ***(Function) Scope Manager***: Got it, here it is.
>
> ***Compiler***: Hey *Scope Manager* (of the function), I found a formal parameter declaration for `studentID`, ever heard of it?
>
> ***(Function) Scope Manager***: Nope, but now it's registered in this scope.
>
> ***Compiler***: Hey *Scope Manager* (of the function), I found a `for`-loop that will need its own scope bucket.
>
> ...

The conversation is a question-and-answer exchange, where **Compiler** asks the current *Scope Manager* if an encountered identifier declaration has already been encountered? If "no", *Scope Manager* creates that variable in that scope. If the answer were "yes", then it would effectively be skipped over since there's nothing more for that *Scope Manager* to do.

Compiler also signals when it runs across functions or block scopes, so that a new scope bucket and *Scope Manager* can be instantiated.

Later, when it comes to execution of the program, the conversation will proceed between *Engine* and *Scope Manager*, and might play out like this:

> ***Engine***: Hey *Scope Manager* (of the global scope), before we begin, can you lookup the identifier `getStudentName` so I can assign this function to it?
>
> ***(Global) Scope Manager***: Yep, here you go.
>
> ***Engine***: Hey *Scope Manager*, I found a *target* reference for `students`, ever heard of it?
>
> ***(Global) Scope Manager***: Yes, it was formally declared for this scope, and it's already been initialized to `undefined`, so it's ready to assign to. Here you go.
>
> ***Engine***: Hey *Scope Manager* (of the global scope), I found a *target* reference for `nextStudent`, ever heard of it?

> ***(Global) Scope Manager***: Yes, it was formally declared for this scope, and it's already been initialized to `undefined`, so it's ready to assign to. Here you go.
>
> ***Engine***: Hey *Scope Manager* (of the global scope), I found a *source* reference for `getStudentName`, ever heard of it?
>
> ***(Global) Scope Manager***: Yes, it was formally declared for this scope. Here you go.
>
> ***Engine***: Great, the value in `getStudentName` is a function, so I'm going to execute it.
>
> ***Engine***: Hey *Scope Manager*, now we need to instantiate the function's scope.
>
> ...

This conversation is another question-and-answer exchange, where *Engine* first asks the current *Scope Manager* to lookup the hoisted `getStudentName` identifier, so as to associate the function with it. *Engine* then proceeds to ask *Scope Manager* about the *target* reference for `students`, and so on.

To review and summarize how a statement like `var students = [ .. ]` is processed, in two distinct steps:

1. *Compiler* sets up the declaration of the scope variable (since it wasn't previously declared in the current scope).
2. While *Engine* is executing, since the declaration has an initialization assignment, *Engine* asks *Scope Manager* to look up the variable, and assigns to it once found.

Nested Scope

When it comes time to execute the `getStudentName()` function, *Engine* asks for a *Scope Manager* instance for that function's scope, and it will then proceed to lookup the parameter (`studentID`) to assign the `73` argument value to, and so on.

The function scope for `getStudentName(..)` is nested inside the global scope. The block scope of the `for`-loop is similarly nested inside that function scope. Scopes can be lexically nested to any arbitrary depth as the program defines.

Each scope gets its own *Scope Manager* instance each time that scope is executed (one or more times). Each scope automatically has all its identifiers registered (this is called "variable hoisting"; see Chapter 3).

At the beginning of a scope, if any identifier came from a `function` declaration, that variable is automatically initialized to its associated function reference. And if any identifier came from a `var` declaration (as opposed to `let` / `const`), that variable is automatically initialized to `undefined` so that it can be used; otherwise, the variable remains uninitialized (aka, in its "TDZ", see Chapter 3) and cannot be used until its declaration-and-initialization are executed.

In the `for (let student of students) {` statement, `students` is a *source* reference that must be looked up. But how will that lookup be handled, since the scope of the function will not find such an identifier.

To understand that, let's imagine that bit of conversation playing out like this:

> ***Engine***: Hey *Scope Manager* (for the function), I have a *source* reference for `students`, ever heard of it?
>
> ***(Function) Scope Manager***: Nope, never heard of it. Try the next outer scope.
>
> ***Engine***: Hey *Scope Manager* (for the global scope), I have a *source* reference for `students`, ever heard of it?
>
> ***(Global) Scope Manager***: Yep, it was formally declared, here you go.
>
> ...

One of the most important aspects of lexical scope is that any time an identifier reference cannot be found in the current scope, the next outer scope in the nesting is consulted; that process is repeated until an answer is found or there are no more scopes to consult.

Lookup Failures

When *Engine* exhausts all *lexically available* scopes and still cannot resolve the lookup of an identifier, an error condition then exists. However, depending on the mode of the program (strict-mode or not) and the role of the variable (i.e., *target* vs. *scope*; see Chapter 1), this error condition will be handled differently.

If the variable is a *source*, an unresolved identifier lookup is considered an undeclared (unknown, missing) variable, which results in a `ReferenceError`

being thrown. Also, if the variable is a *target*, and the code at that point is running in strict-mode, the variable is considered undeclared and throws a `ReferenceError`.

WARNING: |
:— |
The error message for an undeclared variable condition, in most JS environments, will likely say, "Reference Error: XYZ is not defined". The phrase "not defined" seems almost identical to the term "undefined", as far as the English language goes. But these two are very different in JS, and this error message unfortunately creates a likely confusion. "Not defined" really means "not declared", or rather "undeclared", as in a variable that was never formally declared in any *lexically available* scope. By contrast, "undefined" means a variable was found (declared), but the variable otherwise has no value in it at the moment, so it defaults to the `undefined` value. Yes, this terminology mess is confusing and terribly unfortunate. |

However, if the variable is a *target* and strict-mode is not in effect, a confusing and surprising legacy behavior occurs. The extremely unfortunate outcome is that the global scope's *Scope Manager* will just create an **accidental global variable** to fulfill that target assignment!

```
function getStudentName() {
    // assignment to an undeclared variable :(
    nextStudent = "Suzy";
}

getStudentName();

console.log(nextStudent);
// "Suzy" -- oops, an accidental-global variable!
```

Yuck.

This sort of accident (almost certain to lead to bugs eventually) is a great example of the protections of strict-mode, and why it's such a bad idea not to use it. Never rely on accidental global variables like that. Always use strict-mode, and always formally declare your variables. You'll then get a helpful `ReferenceError` if you ever mistakenly try to assign to a not-declared variable.

Building On Metaphors

To visualize nested scope resolution, yet another useful metaphor may be an office building:

Fig. 2: Scope "Building"

The building represents our program's nested scope rule set. The first floor of the building represents the currently executing scope. The top level of the building is the global scope.

You resolve a *target* or *source* variable reference by first looking on the current floor, and if you don't find it, taking the elevator to the next floor, looking there, then the next, and so on. Once you get to the top floor (the global scope), you either find what you're looking for, or you don't. But you have to stop regardless.

Continue The Conversation

By this point, hopefully you feel more solid on what scope is and how the JS engine determines it while compiling your code.

Before *continuing*, go find some code in one of your projects and run through the conversations. If you find yourself confused or tripped up, spend time reviewing this material.

As we move forward, we want to look in much more detail at how we use lexical scope in our programs.

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.

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

Chapter 4: Block Scope

NOTE: |
:— |
Work in progress |

If you made it this far, through that long Chapter 3, you're likely feeling a lot more aware of, and hopefully more comfortable with, the breadth and depth of scopes and their impact on your code.

Now, we want to focus on one specific form of scope: block scope. Just as we're narrowing our discussion focus in this chapter, so too is block scope about narrowing the focus of a larger scope down to a smaller subset of our program.

Least Exposure

It makes sense that functions define their own scopes. But why do we need blocks to create scopes as well?

Software engineering articulates a fundamental pattern, typically applied to software security, called "The Principle of Least Privilege" (POLP, https://en.wikipedia.org/wiki/Principle_of_least_privilege). And a variation of this principle that applies to our current discussion is typically labeled "Least Exposure".

POLP expresses a defensive posture to software architecture: components of the system should be designed to function with least privilege, least access, least exposure. If each piece is connected with minimum-necessary capabilities, the overall system is stronger from a security standpoint, because a compromise or failure of one piece has a minimized impact on the rest of the system.

If PLOP focuses on system-level component design, the *Exposure* variant (POLE) can be focused on a lower level; we'll apply it to our exploration of how scopes interact with each other.

In following POLE, what do we want to minimize the exposure of? The variables registered in each scope.

Think of it this way: why wouldn't you just place all the variables of your program out in the global scope? That probably immediately feels like a bad idea, but it's worth considering why that is. When variables used by one part of the program are exposed to another part of the program, via scope, there are 3 main hazards that can arise:

1. **Naming Collisions**: if you use a common and useful variable/function name in two different parts of the program, but the identifier comes from one shared scope (like the global scope), then name collision occurs, and it's very likely that bugs will occur as one part uses the variable/function in a way the other part doesn't expect. For example, imagine if all your loops used a single global `i` index variable, and then it happened that one loop in a function was run during an iteration of a loop from another function, and now the shared `i` variable has an unexpected value.

2. **Unexpected Behavior**: if you expose variables/functions whose usage is otherwise *private* to a piece of the program, it allows other developers to use them in ways you didn't intend, which can violate expected behavior and cause bugs. For example, if your part of the program assumes an array contains all numbers, but someone else's code accesses and modifies the array to include booleans or strings, your code may then misbehave in unexpected ways.

   Worse, exposure of *private* details invites those with mal-intent to try to work around limitations you have imposed, to do things with your part of the software that shouldn't be allowed.

3. **Unintended Dependency**: if you expose variables/functions unnecessarily, it invites other developers to use and depend on those otherwise *private* pieces. While that doesn't break your program today, it creates a refactoring hazard in the future, because now you cannot as easily refactor that variable or function without potentially breaking other parts of the software that you don't control. For example, If your code relies on an array of numbers, and you later decide it's better to use some other data structure instead of an array, you now must take on the liability of fixing other affected parts of the software.

POLE, as applied to variable/function scoping, essentially says, default to exposing the bare minimum necessary, keeping everything else as private as possible. Declare variables in as small and deeply nested of scopes as possible, rather than placing everything in the global (or even outer function) scope.

If you design your software accordingly, you have a much greater chance of avoiding (or at least minimizing) these 3 hazards.

Consider:

```
function diff(x,y) {
    if (x > y) {
        let tmp = x;
        x = y;
        y = tmp;
    }
```

```
    return y - x;
}

diff(3,7);      // 4
diff(7,5);      // 2
```

In this `diff(..)` function, we want to ensure that `y` is greater than or equal to `x`, so that when we subtract (`y - x`), the result is `0` or larger. If `x` is larger, we swap `x` and `y` using a `tmp` variable.

In this simple example, it doesn't seem to matter whether `tmp` is inside the `if` block or whether it belongs at the function level – but it certainly shouldn't be a global variable! However, following the POLE principle, `tmp` should be as hidden in scope as possible. So we block scope `tmp` (using `let`) to the `if` block.

Hiding In Plain (Function) Scope

It should now be clear why it's important to hide our variable and function declarations in the lowest (most deeply nested) scopes possible. But how do we do so?

You've already seen `let` (and `const`) declarations, which are block scoped declarators, and we'll come back to them in more detail shortly. But what about hiding `var` or `function` declarations in scopes? That can easily be done by wrapping a `function` scope around a declaration.

Let's consider an example where `function` scoping can be useful.

The mathematical operation "factorial" (notated as "6!") is the multiplication of a given integer against all successively lower integers down to `1` – actually, you can stop at `2` since multiplying `1` does nothing. In other words, "6!" is the same as "6 * 5!", which is the same as "6 * 5 * 4!", and so on. Because of the nature of the math involved, once any given integer's factorial (like "4!") has been calculated, we shouldn't need to do that work again, as it'll always be the same answer.

So if you naively calculate factorial for `6`, then later want to calculate factorial for `7`, you might unnecessarily re-calculate the factorials of all the integers from 1 up to 6. If you're willing to trade memory for speed, you can solve that wasted computation by caching each integer's factorial as it's calculated:

```
var cache = {};

function factorial(x) {
    if (x < 2) return 1;
    if (!(x in cache)) {
        cache[x] = x * factorial(x - 1);
```

```
    }
    return cache[x];
}

factorial(6);
// 720

cache;
// {
//     "2": 2,
//     "3": 6,
//     "4": 24,
//     "5": 120,
//     "6": 720
// }

factorial(7);
// 5040
```

We're storing all the computed factorials in `cache` so that across multiple calls to `factorial(..)`, the answers remain. But the `cache` variable is pretty obviously a *private* detail of how `factorial(..)` works, not something that should be exposed in an outer scope – especially not the global scope.

NOTE: |
:— |
`factorial(..)` here is recursive – a call to itself is made from inside – but that's just for brevity of code sake; a non-recursive implementation would yield the same scoping analysis with respect to `cache`. |

However, fixing this over-exposure issue is not as simple as hiding the `cache` variable inside `factorial(..)`, as it might seem. Since we need `cache` to survive multiple calls, it must be located in a scope outside that function. So what can we do?

Define another middle scope (between the outer/global scope and the inside of `factorial(..)`) for `cache` to live in:

```
// outer/global scope

function hideTheCache() {
    // "middle scope", where we hide ‘cache‘
    var cache = {};

    function factorial(x) {
        // inner scope
        if (x < 2) return 1;
```

```
        if (!(x in cache)) {
            cache[x] = x * factorial(x - 1);
        }
        return cache[x];
    }

    return factorial;
}

var factorial = hideTheCache();

factorial(6);
// 720

factorial(7);
// 5040
```

The `hideTheCache()` function serves no other purpose than to create a scope for `cache` to persist in across multiple calls to `factorial(..)`. But for `factorial(..)` to have access to `cache`, we have to define `factorial(..)` inside that same scope. Then we return the function reference, as a value from `hideTheCache()`, and store it back in an outer scope variable, also named `factorial`. Now we can call `factorial(..)` (multiple times!); its persistent `cache` stays hidden but accessible only to `factorial(..)`!

OK, but... it's going to be tedious to define (and name!) a `hideTheCache(..)` function scope each time such a need for variable/function hiding occurs, especially since we'll likely want to avoid name collisions with this function by giving each occurrence a unique name. Ugh.

NOTE: |
:— |
The illustrated technique – caching a function's computed output to optimize performance when repeated calls of the same inputs are expected – is quite common in the Functional Programming (FP) world, canonically referred to as "memoization". FP libraries will provide a utility for memoization of functions, which would take the place of `hideTheCache(..)` above. Memoization is beyond the *scope* (pun intended!) of our discussion. See my "Functional-Light JavaScript" book for more information. |

Rather than defining a new and uniquely named function each time one of those scope-only-for-the-purpose-of-hiding-a-variable situations occurs, a perhaps better solution is to use a function expression:

```
var factorial = (function hideTheCache() {
    var cache = {};
```

```
    function factorial(x) {
        if (x < 2) return 1;
        if (!(x in cache)) {
            cache[x] = x * factorial(x - 1);
        }
        return cache[x];
    }

    return factorial;
})();

factorial(6);
// 720

factorial(7);
// 5040
```

Wait! This is still using a function to create the scope for hiding `cache`, and in this case, the function is still named `hideTheCache`, so how does that solve anything?

Recall from "Function Name Scope" (Chapter 3), what happens to the name identifier from a function expression. Since `hideTheCache(..)` is defined as a `function` expression instead of a `function` declaration, its name is in its own scope – essentially the same scope as `cache` – rather than in the outer/global scope.

That means we could name every single occurrence of such a function expression the exact same name, and never have any collision. More appropriately, we could name each occurrence semantically based on whatever it is we're trying to hide, and not worry that whatever name we choose is going to collide with any other `function` expression scope in the program.

In fact, as we discussed in "Function Name Scope" (Chapter 3), we *could* just leave off the name entirely – thus defining an "anonymous `function` expression" instead. Appendix A will explore the importance of names for such functions.

Invoking Function Expressions Immediately

But there's another important bit in the above program that's easy to miss: the line at the end of the `function` expression that contains `})();`.

Notice that we surrounded the entire `function` expression in a set of `( .. )`, and then on the end, we added that second `()` parentheses set; that's actually calling the `function` expression we just defined. Moreover, in this case, the first set of surrounding `( .. )` around the function expression is not strictly

necessary (more on that in a moment), but we used them for readability sake anyway.

So, in other words, we're defining a `function` expression that's then immediately invoked. This common pattern has a (very creative!) name: Immediately Invoked Function Expression (IIFE).

IIFEs are useful when we want to create a scope to hide variables/functions. Since they are expressions, they can be used in **any** place in a JS program where an expression is allowed. IIFEs can be named, as with `hideTheCache()`, or (much more commonly!) unnamed/anonymous. And they can be standalone or, as above, part of another statement – `hideTheCache()` returns the `factorial()` function reference which is then `=` assigned to a variable `factorial`.

For comparison, here's an example of a standalone (anonymous) IIFE:

```
// outer scope

(function(){

    // inner hidden scope

})();

// more outer scope
```

Unlike earlier with `hideTheCache()`, where the outer surrounding `(..)` were noted as being an optional stylistic choice, for standalone IIFEs, they're **required**, to distinguish the `function` as an expression and not a statement. So for consistency, it's best to always surround IIFE `function`s with `( .. )`.

NOTE: |
:— |
Technically, the surrounding `( .. )` aren't the only syntactic way to ensure a `function` in an IIFE is treated by the JS parser as a function expression. We'll look at some other options in Appendix A. |

Function Boundaries

Be aware that using an IIFE to define a scope can have some unintended consequences, depending on the code around it. Because an IIFE is a full function, the function boundary alters the behavior of certain statements/constructs.

For example, a `return` statement in some piece of code would change its meaning if then wrapped in an IIFE, because now the `return` would refer to the IIFE's function. Non-arrow function IIFEs also change the binding of a `this` keyword.

And statements like `break` and `continue` won't operate across an IIFE function boundary to control an outer loop or block.

So, if the code you need to wrap a scope around has `return`, `this`, `break`, or `continue` in it, an IIFE is probably not the best approach. In that case, you might look to create the scope with a block instead of a function.

Scoping With Blocks

You should by this point feel fairly comfortable with creating scopes to prevent unnecessary identifier exposure.

And so far, we looked at doing this via `function` (i.e., IIFE) scope. But let's now consider using `let` declarations with nested blocks. In general, any `{ .. }` curly-brace pair which is a statement will act as a block, but **not necessarily** as a scope. A block only becomes a scope if it needs to, to contain any block-scoped declarations (i.e., `let` or `const`) present within it.

Consider:

```
{
    let thisIsNowAScope = true;

    for (let i = 0; i < 5; i++) {
        // this is also a scope, activated each iteration

        if (i % 2 == 0) {
            // this is just a block, not a scope
            console.log(i);
        }
    }
}
// 0 2 4
```

NOTE: |
:— |
Not all `{ .. }` curly-brace pairs create blocks (and thus are eligible to become scopes). Object literals use `{ .. }` curly-brace pairs to delimit their key-value lists, but such objects are **not** scopes. `class` uses `{ .. }` curly-braces around its body definition, but this is not a block or scope. A `function` uses `{ .. }` around its body, but this is not technically a block – it's a single statement for the function body – though it *is* a (function) scope. The `{ .. }` curly-brace pair around the `case` clauses of a `switch` statement does not define a block/scope. |

Blocks can be defined as part of a statement (like an `if` or `for`), or as bare standalone block statements, as shown in the outermost `{ .. }` curly brace

pair in the snippet above. An explicit block of this sort, without any declarations (and thus, not actually a scope) serves no operational purpose, though it can be useful as a stylistic signal.

Explicit blocks were always valid JS syntax, but since they couldn't be a scope, they were extremely uncommon prior to ES6's introduction of `let` / `const`. Now they're starting to catch on a little bit.

In most languages that support block scoping, an explicit block scope is a very common pattern for creating a narrow slice of scope for one or a few variables. So following the POLE principle, we should embrace this pattern more widespread in JS as well; use block scoping to narrow the exposure of identifiers to the minimum practical.

An explicit block scope can be useful even inside of another block (whether it's a scope or not).

For example:

```
if (somethingHappened) {
    // this is a block, but not a scope

    {
        // this is an explicit block scope
        let msg = somethingHappened.message();
        notifyOthers(msg);
    }

    recoverFromSomething();
}
```

Here, the `{ .. }` curly-brace pair inside the `if` statement is an even smaller inner explicit block scope for `msg`, since that variable is not needed for the entire `if` block. Most developers would just block-scope `msg` to the `if` block and move on. When there's only a few lines to consider, it's a toss-up judgement call. As code grows, these issues become more pronounced.

So does it matter enough to add the extra `{ .. }` pair and indentation level? I think you should follow POLE and always define the smallest block (within reason!) for each variable. So I would recommend using the extra explicit block scope.

WARNING: |
:— |
Recall the discussion of TDZ errors from "Uninitialized" (Chapter 3). My suggestion to minimize the risk of TDZ errors with `let` (or `const`) declarations is to always put the declarations at the top of any scope. If you find yourself putting a `let` declaration in the middle of a scope block, first think, "Oh, no!

TDZ alert!". Then recognize that if this `let` declaration isn't actually needed for the whole block, you could/should use an inner explicit block scope to further narrow its exposure! |

Another example making use of an explicit block scope:

```
function getNextMonthStart(dateStr) {
    var nextMonth, year;

    {
        let curMonth;
        [ , year, curMonth ] = dateStr.match(/(\d{4})-(\d{2})-\d{2}/) || [];
        nextMonth = (Number(curMonth) % 12) + 1;
    }

    // did we cross a year boundary?
    if (nextMonth == 1) {
        year++;
    }

    return `${ year }-${ String(nextMonth).padStart(2,"0") }-01`;
}

getNextMonthStart("2019-12-25");
// 2020-01-01
```

Let's first identify the scopes and their identifiers:

1. The outer/global scope has one identifier, the function `getNextMonthStart(..)`.
2. The function scope for `getNextMonthStart(..)` has three identifiers: `dateStr` (the parameter), `nextMonth`, and `year`.
3. The `{ .. }` curly-brace pair defines an inner block scope that includes one variable: `curMonth`.

So why did we put `curMonth` in an explicit block scope instead of just alongside `nextMonth` and `year` in the top-level function scope? Because `curMonth` is only needed for those first two statements. Exposing it at the function scope level is, in essence, over-exposing it.

This example is small, so the hazards are pretty minimal in over-exposing the scoping of `curMonth`. But the benefits of the POLE principle are best achieved when you adopt the mindset of minimizing scope exposure as a habit. If you follow the principle consistently even in the small cases, it will serve you more as your programs grow.

Let's now look at an even more substantial example:

```
function sortNamesByLength(names) {
    var buckets = [];

    for (let firstName of names) {
        if (buckets[firstName.length] == null) {
            buckets[firstName.length] = [];
        }
        buckets[firstName.length].push(firstName);
    }

    // a block to narrow the scope
    {
        let sortedNames = [];

        for (let bucket of buckets) {
            if (bucket) {
                // sort each bucket alphanumerically
                bucket.sort();

                // append the sorted names to our running list
                sortedNames = [ ...sortedNames, ...bucket ];
            }
        }

        return sortedNames;
    }
}

sortNamesByLength([
    "Sally",
    "Suzy",
    "Frank",
    "John",
    "Jennifer",
    "Scott"
]);
// [ "John", "Suzy", "Frank", "Sally", "Scott", "Jennifer" ]
```

There are six identifiers declared across five different scopes. Could all of these variables have existed in the single outer/global scope? Technically, yes, since they're all uniquely named and thus have no name collisions. But this would be really poor code organization, and would likely lead to both confusion and future bugs.

We split them out into each inner nested scope as appropriate. Each variable is defined at the innermost scope possible for the program to operate as desired.

`sortedNames` could have been defined in the top-level function scope, but it's only needed for the second half of this function. To avoid over-exposing that variable in a higher level scope, we again follow POLE and block-scope it in the inner block scope.

`var` AND `let`

Next, let's talk about the declaration `var buckets`. That variable is used across the entire function (except the final `return` statement). Any variable that is needed across all (or even most) of a function should be declared so that such usage is obvious.

NOTE: |
:— |
The parameter `names` isn't used across the whole function, but there's no way limit the scope of a parameter, so it behaves as a function-wide declaration regardless. |

So why did we use `var` instead of `let` to declare the `buckets` variable? There's both stylistic and technical reasons to choose `var` here.

Stylistically, `var` has always, from the earliest days of JS, signaled "variable that belongs to a whole function". As we asserted in "Lexical Scope" (Chapter 1), `var` attaches to the nearest enclosing function scope, no matter where it appears. That's true even if `var` appears inside a block:

```
function diff(x,y) {
    if (x > y) {
        var tmp = x;    // `tmp` is function-scoped
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Even though `var` is inside a block, its declaration is function-scoped (to `diff(..)`), not block-scoped.

While you can declare `var` inside a block (and still have it be function-scoped), I would recommend you minimize this approach except in a few specific cases (discussed in Appendix A). Otherwise, `var` should be reserved for use in the top-level scope of a function.

Why not just use `let` in that same location? Because `var` is visually distinct from `let` and therefore signals clearly, "this variable is function-scoped". Using `let` in the top-level scope, especially if not in the first few lines of a function,

especially when all the other declarations in blocks use `let`, does not visually draw attention to the difference with the function-scoped declaration.

In other words, I feel `var` better communicates function-scoped than `let` does, and `let` both communicates (and achieves!) block-scoping where `var` is insufficient. As long as your programs are going to need both function-scoped and block-scoped variables, the most sensible and readable approach is to use both `var` AND `let` together, each for their own best purpose.

There are other stylistic and operational reasons to choose `var` or `let` in different scenarios. We'll explore the case for `var` alongside `let` in more detail in Appendix A.

WARNING: |
:— |
My advice to use `var` AND `let` here is controversial. It's far more common to hear assertions like, "var is broken, let fixes it" and, "never use var, let is the replacement". Those opinions are as valid as my opinions, but they're just opinions; `var` is not factually broken or deprecated. It has worked since early JS and it will continue to always work as long as JS is around. |

When To `let`?

My advice to reserve `var` for (mostly only) a top-level function scope means that all other declarations should use `let`. But you may still be wondering how to decide where each declaration in your program belongs?

POLE already guides you on those decisions, but let's make sure we explicitly state it. The way to decide is not based on which keyword you want to use. The way to decide is to ask, "What is the most minimal scope exposure that's sufficient for this variable?" Once that is answered, you'll know if a variable belongs in a block scope or the function scope. If you decide initially that a variable should be block-scoped, and later realize it needs to be elevated to be function-scoped, then that dictates a change not only in the location of that variable's declaration, but also the keyword used. The decision making process really should proceed like that

If a declaration belongs in a block scope, use `let`. If it belongs in the function scope, use `var` (again, my opinion).

But another way to sort of visualize this decision making is to consider the pre-ES6 version of a program. For example, let's recall `diff(..)` from earlier:

```
function diff(x,y) {
    var tmp;

    if (x > y) {
        tmp = x;
```

```
        x = y;
        y = tmp;
    }

    return y - x;
}
```

In this version of `diff(..)`, `tmp` is clearly declared in the function scope. Is that appropriate for `tmp`? I would argue, no. `tmp` is only needed for those few statements. It's not needed once we get to the `return` statement. It should therefore be block-scoped.

Prior to ES6, we didn't have `let` so we couldn't *actually* block-scope it. But we could do the next-best thing:

```
function diff(x,y) {
    if (x > y) {
        // ‘tmp‘ is still function-scoped, but
        // it signals block-scoping stylistically
        var tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Placing the `var` declaration for `tmp` inside the `if` statement signals to the reader of the code that `tmp` belongs to that block. Even though JS doesn't enforce that scoping, the stylistic signal still has benefit for the reader of your code.

Now, you can just locate any `var` that's inside a block of this sort and switch it to `let` to enforce the signal already being sent stylistically.

Another example that used to be commonly based on `var` but which should pretty much always be `let` is a `for` loop:

```
for (var i = 0; i < 5; i++) {
    // do something
}
```

No matter where such a loop is defined, the `i` should basically always be used only inside the loop, in which case POLE tells us it should be declared with `let` instead of `var`:

```
for (let i = 0; i < 5; i++) {
    // do something
}
```

Basically the only time switching a `var` to a `let` in this way would "break" your code is if you were relying on accessing the loop's iterator (`i`) outside/after the loop, such as:

```
for (var i = 0; i < 5; i++) {
    if (checkValue(i)) {
        break;
    }
}

if (i < 5) {
    console.log("The loop stopped early!");
}
```

This usage pattern is not terribly uncommon, but most feel it signals poor code structure. A better approach is to introduce another outer-scoped variable:

```
var lastI;

for (let i = 0; i < 5; i++) {
    lastI = i;
    if (checkValue(i)) {
        break;
    }
}

if (lastI < 5) {
    console.log("The loop stopped early!");
}
```

`lastI` is needed across this whole scope (function or global), so it's declared with `var`. `i` is only needed in (each) loop iteration, so it's declared with `let`.

What's The Catch?

So far we've asserted that `var` and parameters are function-scoped, and `let` / `const` signal block-scoped declarations. There's one little exception to call out: the `catch` clause.

Since the introduction of `try..catch` way back in ES3, the `catch` clause has held an additional (little-known) block-scoping declaration capability:

```
try {
    doesntExist();
}
catch (err) {
    console.log(err);
    // ReferenceError: 'doesntExist' is not defined
    // ^^^^ this is printed from the caught exception

    let onlyHere = true;
    var outerVariable = true;
}

console.log(outerVariable);     // true

console.log(err);
// ReferenceError: 'err' is not defined
// ^^^^ this is another thrown (uncaught) exception
```

The `err` variable declared by the `catch` clause is block-scoped to that block. This clause block can have other block-scoped declarations via `let`. But a `var` declaration inside this block still attaches to the outer function/global scope.

ES2019 (recently, at the time of writing) changed `catch` clauses so their declaration is optional. If you need to catch *that an exception occurred* (so you can gracefully recover), but you don't care about the error value itself, you can omit that declaration:

```
try {
    doOptionOne();
}
catch {  // catch-declaration omitted
    doOptionTwoInstead();
}
```

NOTE: |
:— |
In this case, the `catch` block is **not** a scope unless a `let` declaration is added inside it. |

This is a small but delightful simplification of syntax for a fairly common use-case!

Functions Declarations In Blocks (FiB)

We've seen now that `let` / `const` declarations are block-scoped, and `var` declarations are function-scoped. So what about `function` declarations that appear directly inside blocks (as a feature, called "FiB")?

We typically think of `function` declarations like they're the equivalent of a `var` declaration. So are they function-scoped?

No, and yes. I know... that's confusing. Let's dig in.

Consider:

```
if (false) {
    function ask() {
        console.log("Does this run?");
    }
}

ask();
```

What do you expect for this program to do? Three reasonable outcomes:

1. The `ask()` call might fail with a Reference Error exception, because the `ask` identifier is block-scoped to the `if` block scope and thus isn't available in the outer/global scope.
2. The `ask()` call might fail with a Type Error exception, because the `ask` identifier exists, but it's `undefined` (since the `if` statement doesn't run) and thus not a callable function.
3. The `ask()` call might run correctly, printing out the "Does it run?" message.

Here's the confusing part: depending on which JS environment you try that code snippet in, you may get different results! This is one of those few crazy areas where existing legacy behavior interferes with getting a predictable outcome.

The JS specification says that `function` declarations inside of blocks are block-scoped, so the answer should be (1). However, most browser-based JS engines (including v8 which comes from Chrome but is also used in Node) will behave as (2), meaning the identifier is scoped outside the `if` block but the function value is not automatically initialized, so it remains `undefined`.

Why are browser JS engines allowed to behave contrary to the specification? Because these engines already had certain behaviors around FiB before ES6 introduced block scoping, and there was concern that changing to adhere to the specification might break some existing website JS code. As such, an exception was made in Appendix B of the specification, which allows certain deviations for browser JS engines (only!).

NOTE: |
:— |
You wouldn't typically categorize Node as being a browser JS environment, since

it usually runs on a server. But it's an interesting corner case since it shares the v8 engine with the Chrome (and Edge, now) browsers. Since v8 is first a browser JS engine, it follows this Appendix B exception, which then means that the browser exceptions are extended to Node. |

One of the most common use-cases for placing a `function` declaration in a block is to conditionally define a function one way or another (like with an `if..else` statement) depending on some environment state. For example:

```
if (typeof Array.isArray != "undefined") {
    function isArray(a) {
        return Array.isArray(a);
    }
}
else {
    function isArray(a) {
        return Object.prototype.toString.call(a) == "[object Array]";
    }
}
```

It's tempting to structure code this way for performance reasons, since the `typeof Array.isArray` check is only performed once, as opposed to defining just one `isArray(..)` and putting the `if` statement inside it, where the check then runs unnecessarily on every call.

WARNING: |
:— |
In addition to the risks of FiB deviations, one problem with the conditional-definition of functions is that it is harder to debug such a program. If you end up with a bug in the `isArray(..)` function, you first have to figure out *which* `isArray(..)` function definition is actually applied! Sometimes, the bug is that the wrong one got applied because the conditional check was incorrect! If you allow a program to define multiple versions of a function, that program is always harder to reason about and maintain. |

In addition to the situations in the above snippets, there are several other corner cases around FiB you can be bitten by; such behaviors in various browsers and non-browser JS environments (that is, JS engines that aren't primarily browser based) will likely vary.

For example:

```
if (true) {
    function ask() {
        console.log("Am I called?");
    }
}
```

```
if (true) {
    function ask() {
        console.log("Or what about me?");
    }
}

for (let i = 0; i < 5; i++) {
    function ask() {
        console.log("Or is it one of these?");
    }
}

ask();

function ask() {
    console.log("Maybe, it's this one?");
}
```

Recall that function hoisting as described in "When Can I Use A Variable?" (Chapter 3) might suggest that the final `ask()` in this snippet, with "Maybe..." as its message, would hoist above the call to `ask()`. Since it's the last function declaration of that name, it should "win", right? Unfortunately, no.

It's not my intention to document all these weird corner cases, nor to try to explain why each of them behaves a certain way. That information is, in my opinion, useless legacy trivia.

My real concern with FiB is, what advice can I give to ensure your code behaves predictably in all circumstances?

As far as I'm concerned, the only practical answer to avoiding the vagaries of FiB is to simply avoid FiB entirely. In other words, never place a `function` declaration directly inside any block. Always place `function` declarations at the top-level scope of a function (or in the global scope).

So for the earlier `if..else` example, my suggestion is to avoid conditionally defining functions if at all possible. Yes it may be slightly less performant, but this is a better overall option in my opinion:

```
function isArray(a) {
    if (typeof Array.isArray != "undefined") {
        return Array.isArray(a);
    }
    else {
        return Object.prototype.toString.call(a) == "[object Array]";
    }
}
```

If that performance hit becomes a critical path issue for your application, I suggest you consider this approach:

```
var isArray = function isArray(a) {
    return Array.isArray(a);
};

// override the definition, if you must
if (typeof Array.isArray == "undefined") {
    isArray = function isArray(a) {
        return Object.prototype.toString.call(a) == "[object Array]";
    };
}
```

It's important to notice that here I'm placing a `function` **expression**, not a declaration, inside the `if` statement. That's perfectly fine and valid, for `function` expressions to appear inside blocks. Our discussion about FiB is about avoiding `function` **declarations** in blocks.

Even if you test your program and it works correctly, the small benefit you may derive from using FiB style in your code is far outweighed by the potential risks in the future for confusion by other developers, or variances in how your code runs in other JS environments.

FiB is not worth it, and should be avoided.

Blocked Over

The whole point of lexical scoping rules in a programming language is so we can appropriately organize our program's variables, both for operational as well as stylistic code communication purposes.

And one of the most important organizational techniques we can learn to get better at is to ensure that no variable is over-exposed to unnecessary scopes (POLE). Hopefully you now appreciate block scoping much more deeply than before.

This whole discussion of lexical scoping so far in the book is intended to build up a solid foundation of understanding so we can tackle the topic of closure effectively. Before moving on to the next chapter, make sure you are fully comfortable with all the ins and outs of lexical scope.

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

Chapter 5: Closures

NOTE: |
:— |
Work in progress |

.

.

.

.

.

.

.

NOTE: |
:— |
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. Please ignore. |

We arrive at this point with hopefully a very healthy, solid understanding of how scope works.

We turn our attention to an incredibly important, but persistently elusive, *almost mythological*, part of the language: **closure**. If you have followed our discussion of lexical scope thus far, the payoff is that closure is going to be, largely, anticlimactic, almost self-obvious. *There's a man behind the wizard's curtain, and we're about to see him.* No, his name is not Crockford!

If however you have nagging questions about lexical scope, now would be a good time to go back and review Chapter 2 before proceeding.

Enlightenment

For those who are somewhat experienced in JavaScript, but have perhaps never fully grasped the concept of closures, *understanding closure* can seem like a special nirvana that one must strive and sacrifice to attain.

I recall years back when I had a firm grasp on JavaScript, but had no idea what closure was. The hint that there was *this other side* to the language, one which promised even more capability than I already possessed, teased and taunted me. I remember reading through the source code of early frameworks trying to understand how it actually worked. I remember the first time something of the "module pattern" began to emerge in my mind. I remember the *a-ha!* moments quite vividly.

What I didn't know back then, what took me years to understand, and what I hope to impart to you presently, is this secret: **closure is all around you in JavaScript, you just have to recognize and embrace it.** Closures are not a special opt-in tool that you must learn new syntax and patterns for. No, closures are not even a weapon that you must learn to wield and master as Luke trained in The Force.

Closures happen as a result of writing code that relies on lexical scope. They just happen. You do not even really have to intentionally create closures to take advantage of them. Closures are created and used for you all over your code. What you are *missing* is the proper mental context to recognize, embrace, and leverage closures for your own will.

The enlightenment moment should be: **oh, closures are already occurring all over my code, I can finally *see* them now.** Understanding closures is like when Neo sees the Matrix for the first time.

Nitty Gritty

OK, enough hyperbole and shameless movie references.

Here's a down-n-dirty definition of what you need to know to understand and recognize closures:

> Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

Let's jump into some code to illustrate that definition.

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a ); // 2
    }

    bar();
}
```

```
foo();
```

This code should look familiar from our discussions of Nested Scope. Function `bar()` has *access* to the variable `a` in the outer enclosing scope because of lexical scope look-up rules (in this case, it's an RHS reference look-up).

Is this "closure"?

Well, technically... *perhaps*. But by our what-you-need-to-know definition above... *not exactly*. I think the most accurate way to explain `bar()` referencing `a` is via lexical scope look-up rules, and those rules are *only* (an important!) **part** of what closure is.

From a purely academic perspective, what is said of the above snippet is that the function `bar()` has a *closure* over the scope of `foo()` (and indeed, even over the rest of the scopes it has access to, such as the global scope in our case). Put slightly differently, it's said that `bar()` closes over the scope of `foo()`. Why? Because `bar()` appears nested inside of `foo()`. Plain and simple.

But, closure defined in this way is not directly *observable*, nor do we see closure *exercised* in that snippet. We clearly see lexical scope, but closure remains sort of a mysterious shifting shadow behind the code.

Let us then consider code which brings closure into full light:

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a );
    }

    return bar;
}

var baz = foo();

baz(); // 2 -- Whoa, closure was just observed, man.
```

The function `bar()` has lexical scope access to the inner scope of `foo()`. But then, we take `bar()`, the function itself, and pass it *as* a value. In this case, we `return` the function object itself that `bar` references.

After we execute `foo()`, we assign the value it returned (our inner `bar()` function) to a variable called `baz`, and then we actually invoke `baz()`, which of course is invoking our inner function `bar()`, just by a different identifier reference.

`bar()` is executed, for sure. But in this case, it's executed *outside* of its declared lexical scope.

After `foo()` executed, normally we would expect that the entirety of the inner scope of `foo()` would go away, because we know that the *Engine* employs a *Garbage Collector* that comes along and frees up memory once it's no longer in use. Since it would appear that the contents of `foo()` are no longer in use, it would seem natural that they should be considered *gone*.

But the "magic" of closures does not let this happen. That inner scope is in fact *still* "in use", and thus does not go away. Who's using it? **The function `bar()` itself**.

By virtue of where it was declared, `bar()` has a lexical scope closure over that inner scope of `foo()`, which keeps that scope alive for `bar()` to reference at any later time.

`bar()` still has a reference to that scope, and that reference is called closure.

So, a few microseconds later, when the variable `baz` is invoked (invoking the inner function we initially labeled `bar`), it duly has *access* to author-time lexical scope, so it can access the variable `a` just as we'd expect.

The function is being invoked well outside of its author-time lexical scope. **Closure** lets the function continue to access the lexical scope it was defined in at author-time.

Of course, any of the various ways that functions can be *passed around* as values, and indeed invoked in other locations, are all examples of observing/exercising closure.

```
function foo() {
    var a = 2;

    function baz() {
        console.log( a ); // 2
    }

    bar( baz );
}

function bar(fn) {
    fn(); // look ma, I saw closure!
}
```

We pass the inner function `baz` over to `bar`, and call that inner function (labeled `fn` now), and when we do, its closure over the inner scope of `foo()` is observed, by accessing `a`.

These passings-around of functions can be indirect, too.

```
var fn;

function foo() {
    var a = 2;

    function baz() {
        console.log( a );
    }

    fn = baz; // assign ‘baz‘ to global variable
}

function bar() {
    fn(); // look ma, I saw closure!
}

foo();

bar(); // 2
```

Whatever facility we use to *transport* an inner function outside of its lexical scope, it will maintain a scope reference to where it was originally declared, and wherever we execute it, that closure will be exercised.

Now I Can See

The previous code snippets are somewhat academic and artificially constructed to illustrate *using closure*. But I promised you something more than just a cool new toy. I promised that closure was something all around you in your existing code. Let us now *see* that truth.

```
function wait(message) {

    setTimeout( function timer(){
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

We take an inner function (named `timer`) and pass it to `setTimeout(..)`. But `timer` has a scope closure over the scope of `wait(..)`, indeed keeping and using a reference to the variable `message`.

A thousand milliseconds after we have executed `wait(..)`, and its inner scope should otherwise be long gone, that inner function `timer` still has closure over that scope.

Deep down in the guts of the *Engine*, the built-in utility `setTimeout(..)` has reference to some parameter, probably called `fn` or `func` or something like that. *Engine* goes to invoke that function, which is invoking our inner `timer` function, and the lexical scope reference is still intact.

Closure.

Or, if you're of the jQuery persuasion (or any JS framework, for that matter):

```
function setupBot(name,selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

I am not sure what kind of code you write, but I regularly write code which is responsible for controlling an entire global drone army of closure bots, so this is totally realistic!

(Some) joking aside, essentially *whenever* and *wherever* you treat functions (which access their own respective lexical scopes) as first-class values and pass them around, you are likely to see those functions exercising closure. Be that timers, event handlers, Ajax requests, cross-window messaging, web workers, or any of the other asynchronous (or synchronous!) tasks, when you pass in a *callback function*, get ready to sling some closure around!

Note: Chapter 3 introduced the IIFE pattern. While it is often said that IIFE (alone) is an example of observed closure, I would somewhat disagree, by our definition above.

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

This code "works", but it's not strictly an observation of closure. Why? Because the function (which we named "IIFE" here) is not executed outside its lexical

scope. It's still invoked right there in the same scope as it was declared (the enclosing/global scope that also holds `a`). `a` is found via normal lexical scope look-up, not really via closure.

While closure might technically be happening at declaration time, it is *not* strictly observable, and so, as they say, *it's a tree falling in the forest with no one around to hear it.*

Though an IIFE is not *itself* an example of closure, it absolutely creates scope, and it's one of the most common tools we use to create scope which can be closed over. So IIFEs are indeed heavily related to closure, even if not exercising closure themselves.

Put this book down right now, dear reader. I have a task for you. Go open up some of your recent JavaScript code. Look for your functions-as-values and identify where you are already using closure and maybe didn't even know it before.

I'll wait.

Now... you see!

Loops + Closure

The most common canonical example used to illustrate closure involves the humble for-loop.

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

Note: Linters often complain when you put functions inside of loops, because the mistakes of not understanding closure are **so common among developers**. We explain how to do so properly here, leveraging the full power of closure. But that subtlety is often lost on linters and they will complain regardless, assuming you don't *actually* know what you're doing.

The spirit of this code snippet is that we would normally *expect* for the behavior to be that the numbers "1", "2", .. "5" would be printed out, one at a time, one per second, respectively.

In fact, if you run this code, you get "6" printed out 5 times, at the one-second intervals.

Huh?

Firstly, let's explain where `6` comes from. The terminating condition of the loop is when `i` is *not* `<=5`. The first time that's the case is when `i` is 6. So, the output is reflecting the final value of the `i` after the loop terminates.

This actually seems obvious on second glance. The timeout function callbacks are all running well after the completion of the loop. In fact, as timers go, even if it was `setTimeout(.., 0)` on each iteration, all those function callbacks would still run strictly after the completion of the loop, and thus print `6` each time.

But there's a deeper question at play here. What's *missing* from our code to actually have it behave as we semantically have implied?

What's missing is that we are trying to *imply* that each iteration of the loop "captures" its own copy of `i`, at the time of the iteration. But, the way scope works, all 5 of those functions, though they are defined separately in each loop iteration, all **are closed over the same shared global scope**, which has, in fact, only one `i` in it.

Put that way, *of course* all functions share a reference to the same `i`. Something about the loop structure tends to confuse us into thinking there's something else more sophisticated at work. There is not. There's no difference than if each of the 5 timeout callbacks were just declared one right after the other, with no loop at all.

OK, so, back to our burning question. What's missing? We need more ~~cowbell~~ closured scope. Specifically, we need a new closured scope for each iteration of the loop.

We learned in Chapter 3 that the IIFE creates scope by declaring a function and immediately executing it.

Let's try:

```
for (var i=1; i<=5; i++) {
    (function(){
        setTimeout( function timer(){
            console.log( i );
        }, i*1000 );
    })();
}
```

Does that work? Try it. Again, I'll wait.

I'll end the suspense for you. **Nope.** But why? We now obviously have more lexical scope. Each timeout function callback is indeed closing over its own per-iteration scope created respectively by each IIFE.

It's not enough to have a scope to close over **if that scope is empty**. Look closely. Our IIFE is just an empty do-nothing scope. It needs *something* in it to be useful to us.

It needs its own variable, with a copy of the `i` value at each iteration.

```
for (var i=1; i<=5; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })();
}
```

Eureka! It works!

A slight variation some prefer is:

```
for (var i=1; i<=5; i++) {
    (function(j){
        setTimeout( function timer(){
            console.log( j );
        }, j*1000 );
    })( i );
}
```

Of course, since these IIFEs are just functions, we can pass in `i`, and we can call it `j` if we prefer, or we can even call it `i` again. Either way, the code works now.

The use of an IIFE inside each iteration created a new scope for each iteration, which gave our timeout function callbacks the opportunity to close over a new scope for each iteration, one which had a variable with the right per-iteration value in it for us to access.

Problem solved!

Block Scoping Revisited

Look carefully at our analysis of the previous solution. We used an IIFE to create new scope per-iteration. In other words, we actually *needed* a per-iteration **block scope**. Chapter 3 showed us the `let` declaration, which hijacks a block and declares a variable right there in the block.

It essentially turns a block into a scope that we can close over. So, the following awesome code "just works":

```
for (var i=1; i<=5; i++) {
    let j = i; // yay, block-scope for closure!
    setTimeout( function timer(){
```

```
        console.log( j );
    }, j*1000 );
}
```

But, that's not all! (in my best Bob Barker voice). There's a special behavior defined for `let` declarations used in the head of a for-loop. This behavior says that the variable will be declared not just once for the loop, **but each iteration**. And, it will, helpfully, be initialized at each subsequent iteration with the value from the end of the previous iteration.

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer(){
        console.log( i );
    }, i*1000 );
}
```

How cool is that? Block scoping and closure working hand-in-hand, solving all the world's problems. I don't know about you, but that makes me a happy JavaScripter.

Review (TL;DR)

Closure seems to the un-enlightened like a mystical world set apart inside of JavaScript which only the few bravest souls can reach. But it's actually just a standard and almost obvious fact of how we write code in a lexically scoped environment, where functions are values and can be passed around at will.

Closure is when a function can remember and access its lexical scope even when it's invoked outside its lexical scope.

Closures can trip us up, for instance with loops, if we're not careful to recognize them and how they work. But they are also an immensely powerful tool, enabling patterns like *modules* in their various forms.

Modules require two key characteristics: 1) an outer wrapping function being invoked, to create the enclosing scope 2) the return value of the wrapping function must include reference to at least one inner function that then has closure over the private inner scope of the wrapper.

Now we can see closures all around our existing code, and we have the ability to recognize and leverage them to our own benefit!

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

Chapter 6: Module Pattern

NOTE: |
:— |
Work in progress |

.

.

.

.

.

.

.

NOTE: |
:— |
Everything below here is previous text from 1st edition, and is only here for reference while 2nd edition work is underway. Please ignore. |

Modules

There are other code patterns which leverage the power of closure but which do not on the surface appear to be about callbacks. Let's examine the most powerful of them: *the module*.

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
```

```
    }
}
```

As this code stands right now, there's no observable closure going on. We simply have some private data variables `something` and `another`, and a couple of inner functions `doSomething()` and `doAnother()`, which both have lexical scope (and thus closure!) over the inner scope of `foo()`.

But now consider:

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

This is the pattern in JavaScript we call *module*. The most common way of implementing the module pattern is often called "Revealing Module", and it's the variation we present here.

Let's examine some things about this code.

Firstly, `CoolModule()` is just a function, but it *has to be invoked* for there to be a module instance created. Without the execution of the outer function, the creation of the inner scope and the closures would not occur.

Secondly, the `CoolModule()` function returns an object, denoted by the object-literal syntax `{ key: value, ... }`. The object we return has references on it to our inner functions, but *not* to our inner data variables. We keep those

hidden and private. It's appropriate to think of this object return value as essentially a **public API for our module**.

This object return value is ultimately assigned to the outer variable `foo`, and then we can access those property methods on the API, like `foo.doSomething()`.

Note: It is not required that we return an actual object (literal) from our module. We could just return back an inner function directly. jQuery is actually a good example of this. The `jQuery` and `$` identifiers are the public API for the jQuery "module", but they are, themselves, just a function (which can itself have properties, since all functions are objects).

The `doSomething()` and `doAnother()` functions have closure over the inner scope of the module "instance" (arrived at by actually invoking `CoolModule()`). When we transport those functions outside of the lexical scope, by way of property references on the object we return, we have now set up a condition by which closure can be observed and exercised.

To state it more simply, there are two "requirements" for the module pattern to be exercised:

1. There must be an outer enclosing function, and it must be invoked at least once (each time creates a new module instance).
2. The enclosing function must return back at least one inner function, so that this inner function has closure over the private scope, and can access and/or modify that private state.

An object with a function property on it alone is not *really* a module. An object which is returned from a function invocation which only has data properties on it and no closured functions is not *really* a module, in the observable sense.

The code snippet above shows a standalone module creator called `CoolModule()` which can be invoked any number of times, each time creating a new module instance. A slight variation on this pattern is when you only care to have one instance, a "singleton" of sorts:

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
```

```
    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Here, we turned our module function into an IIFE (see Chapter 3), and we *immediately* invoked it and assigned its return value directly to our single module instance identifier `foo`.

Modules are just functions, so they can receive parameters:

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

Another slight but powerful variation on the module pattern is to name the object you are returning as your public API:

```
var foo = (function CoolModule(id) {
    function change() {
        // modifying the public API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }
```

```
    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

By retaining an inner reference to the public API object inside your module instance, you can modify that module instance **from the inside**, including adding and removing methods, properties, *and* changing their values.

Modern Modules

Various module dependency loaders/managers essentially wrap up this pattern of module definition into a friendly API. Rather than examine any one particular library, let me present a *very simple* proof of concept **for illustration purposes (only)**:

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }

    return {
        define: define,
        get: get
    };
```

```
})();
```

The key part of this code is `modules[name] = impl.apply(impl, deps)`. This is invoking the definition wrapper function for a module (passing in any dependencies), and storing the return value, the module's API, into an internal list of modules tracked by name.

And here's how I might use it to define some modules:

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Both the "foo" and "bar" modules are defined with a function that returns a public API. "foo" even receives the instance of "bar" as a dependency parameter, and can use it accordingly.

Spend some time examining these code snippets to fully understand the power of closures put to use for our own good purposes. The key take-away is that

there's not really any particular "magic" to module managers. They fulfill both characteristics of the module pattern I listed above: invoking a function definition wrapper, and keeping its return value as the API for that module.

In other words, modules are just modules, even if you put a friendly wrapper tool on top of them.

Future Modules

ES6 adds first-class syntax support for the concept of modules. When loaded via the module system, ES6 treats a file as a separate module. Each module can both import other modules or specific API members, as well export their own public API members.

Note: Function-based modules aren't a statically recognized pattern (something the compiler knows about), so their API semantics aren't considered until run-time. That is, you can actually modify a module's API during the run-time (see earlier `publicAPI` discussion).

By contrast, ES6 Module APIs are static (the APIs don't change at run-time). Since the compiler knows *that*, it can (and does!) check during (file loading and) compilation that a reference to a member of an imported module's API *actually exists*. If the API reference doesn't exist, the compiler throws an "early" error at compile-time, rather than waiting for traditional dynamic run-time resolution (and errors, if any).

ES6 modules **do not** have an "inline" format, they must be defined in separate files (one per module). The browsers/engines have a default "module loader" (which is overridable, but that's well-beyond our discussion here) which synchronously loads a module file when it's imported.

Consider:

bar.js

```
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;
```

foo.js

```
// import only `hello()` from the "bar" module
import hello from "bar";

var hungry = "hippo";
```

```
function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;

// import the entire "foo" and "bar" modules
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Note: Separate files **"foo.js"** and **"bar.js"** would need to be created, with the contents as shown in the first two snippets, respectively. Then, your program would load/import those modules to use them, as shown in the third snippet.

`import` imports one or more members from a module's API into the current scope, each to a bound variable (`hello` in our case). `module` imports an entire module API to a bound variable (`foo`, `bar` in our case). `export` exports an identifier (variable, function) to the public API for the current module. These operators can be used as many times in a module's definition as is necessary.

The contents inside the *module file* are treated as if enclosed in a scope closure, just like with the function-closure modules seen earlier.

Review (TL;DR)

Closure seems to the un-enlightened like a mystical world set apart inside of JavaScript which only the few bravest souls can reach. But it's actually just a standard and almost obvious fact of how we write code in a lexically scoped environment, where functions are values and can be passed around at will.

Closure is when a function can remember and access its lexical scope even when it's invoked outside its lexical scope.

Closures can trip us up, for instance with loops, if we're not careful to recognize them and how they work. But they are also an immensely powerful tool, enabling patterns like *modules* in their various forms.

Modules require two key characteristics: 1) an outer wrapping function being invoked, to create the enclosing scope 2) the return value of the wrapping function

must include reference to at least one inner function that then has closure over the private inner scope of the wrapper.

Now we can see closures all around our existing code, and we have the ability to recognize and leverage them to our own benefit!