You Don't Know JS: Scope & Closures

Chapter 4: Hoisting

By now, you should be fairly comfortable with the idea of scope, and how variables are attached to different levels of scope depending on where and how they are declared. Both function scope and block scope behave by the same rules in this regard: any variable declared within a scope is attached to that scope.

But there's a subtle detail of how scope attachment works with declarations that appear in various locations within a scope, and that detail is what we will examine here.

Chicken Or The Egg?

There's a temptation to think that all of the code you see in a JavaScript program is interpreted line-by-line, top-down in order, as the program executes. While that is substantially true, there's one part of that assumption which can lead to incorrect thinking about your program.

Consider this code:

```
a = 2;

var a;

console.log( a );
```

What do you expect to be printed in the `console.log(..)` statement?

Many developers would expect `undefined`, since the `var a` statement comes after the `a = 2`, and it would seem natural to assume that the variable is re-defined, and thus assigned the default `undefined`. However, the output will be `2`.

Consider another piece of code:

```
console.log( a );

var a = 2;
```

You might be tempted to assume that, since the previous snippet exhibited some less-than-top-down looking

behavior, perhaps in this snippet, `2` will also be printed. Others may think that since the `a` variable is used before it is declared, this must result in a `ReferenceError` being thrown.

Unfortunately, both guesses are incorrect. `undefined` is the output.

So, what's going on here? It would appear we have a chicken-and-the-egg question. Which comes first, the declaration ("egg"), or the assignment ("chicken")?

The Compiler Strikes Again

To answer this question, we need to refer back to Chapter 1, and our discussion of compilers. Recall that the *Engine* actually will compile your JavaScript code before it interprets it. Part of the compilation phase was to find and associate all declarations with their appropriate scopes. Chapter 2 showed us that this is the heart of Lexical Scope.

So, the best way to think about things is that all declarations, both variables and functions, are processed first, before any part of your code is executed.

When you see `var a = 2;`, you probably think of that as one statement. But JavaScript actually thinks of it as two statements: `var a;` and `a = 2;`. The first statement, the declaration, is processed during the compilation phase. The second statement, the assignment, is left **in place** for the execution phase.

Our first snippet then should be thought of as being handled like this:

```
var a;
```

```
a = 2;

console.log( a );
```

...where the first part is the compilation and the second part is the execution.

Similarly, our second snippet is actually processed as:

```
var a;
```

```
console.log( a );

a = 2;
```

So, one way of thinking, sort of metaphorically, about this process, is that variable and function declarations are

"moved" from where they appear in the flow of the code to the top of the code. This gives rise to the name "Hoisting".

In other words, **the egg (declaration) comes before the chicken (assignment)**.

Note: Only the declarations themselves are hoisted, while any assignments or other executable logic are left *in place*. If hoisting were to re-arrange the executable logic of our code, that could wreak havoc.

```
foo();

function foo() {
    console.log( a ); // undefined

    var a = 2;
}
```

The function `foo`'s declaration (which in this case *includes* the implied value of it as an actual function) is hoisted, such that the call on the first line is able to execute.

It's also important to note that hoisting is **per-scope**. So while our previous snippets were simplified in that they only included global scope, the `foo(..)` function we are now examining itself exhibits that `var a` is hoisted to the top of `foo(..)` (not, obviously, to the top of the program). So the program can perhaps be more accurately interpreted like this:

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
}

foo();
```

Function declarations are hoisted, as we just saw. But function expressions are not.

```
foo(); // not ReferenceError, but TypeError!

var foo = function bar() {
    // ...
};
```

The variable identifier `foo` is hoisted and attached to the enclosing scope (global) of this program, so

`foo()` doesn't fail as a `ReferenceError`. But `foo` has no value yet (as it would if it had been a true function declaration instead of expression). So, `foo()` is attempting to invoke the `undefined` value, which is a `TypeError` illegal operation.

Also recall that even though it's a named function expression, the name identifier is not available in the enclosing scope:

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

This snippet is more accurately interpreted (with hoisting) as:

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

Functions First

Both function declarations and variable declarations are hoisted. But a subtle detail (that *can* show up in code with multiple "duplicate" declarations) is that functions are hoisted first, and then variables.

Consider:

```
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

foo = function() {
    console.log( 2 );
};
```

`1` is printed instead of `2` ! This snippet is interpreted by the *Engine* as:

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
    console.log( 2 );
};
```

Notice that `var foo` was the duplicate (and thus ignored) declaration, even though it came before the `function foo()...` declaration, because function declarations are hoisted before normal variables.

While multiple/duplicate `var` declarations are effectively ignored, subsequent function declarations *do* override previous ones.

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

While this all may sound like nothing more than interesting academic trivia, it highlights the fact that duplicate definitions in the same scope are a really bad idea and will often lead to confusing results.

Function declarations that appear inside of normal blocks typically hoist to the enclosing scope, rather than being conditional as this code implies:

```
foo(); // "b"

var a = true;
if (a) {
   function foo() { console.log( "a" ); }
}
else {
   function foo() { console.log( "b" ); }
}
```

However, it's important to note that this behavior is not reliable and is subject to change in future versions of JavaScript, so it's probably best to avoid declaring functions in blocks.

Review (TL;DR)

We can be tempted to look at `var a = 2;` as one statement, but the JavaScript *Engine* does not see it that way. It sees `var a` and `a = 2` as two separate statements, the first one a compiler-phase task, and the second one an execution-phase task.

What this leads to is that all declarations in a scope, regardless of where they appear, are processed *first* before the code itself is executed. You can visualize this as declarations (variables and functions) being "moved" to the top of their respective scopes, which we call "hoisting".

Declarations themselves are hoisted, but assignments, even assignments of function expressions, are *not* hoisted.

Be careful about duplicate declarations, especially mixed between normal var declarations and function declarations -- peril awaits if you do!