You Don't Know JS: Scope & Closures

Appendix B: Polyfilling Block Scope

In Chapter 3, we explored Block Scope. We saw that `with` and the `catch` clause are both tiny examples of block scope that have existed in JavaScript since at least the introduction of ES3.
But it's ES6's introduction of `let` that finally gives full, unfettered block-scoping capability to our code. There are many exciting things, both functionally and code-stylistically, that block scope will enable.
But what if we wanted to use block scope in pre-ES6 environments?

Consider this code:

```
{
        let a = 2;
        console.log( a ); // 2
}

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

This will work great in ES6 environments. But can we do so pre-ES6? `catch` is the answer.

```
try{throw 2}catch(a){
        console.log( a ); // 2
}

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

Whoa! That's some ugly, weird looking code. We see a `try/catch` that appears to forcibly throw an error, but the "error" it throws is just a value `2`, and then the variable declaration that receives it is in the `catch(a)` clause. Mind: blown.
That's right, the `catch` clause has block-scoping to it, which means it can be used as a polyfill for block scope in pre-ES6 environments.
"But...", you say. "...no one wants to write ugly code like that!" That's true. No one writes (some of) the code output by the CoffeeScript compiler, either. That's not the point.

The point is that tools can transpile ES6 code to work in pre-ES6 environments. You can write code using block-scoping, and benefit from such functionality, and let a build-step tool take care of producing code that will actually *work* when deployed.

This is actually the preferred migration path for all (ahem, most) of ES6: to use a code transpiler to take ES6 code and produce ES5-compatible code during the transition from pre-ES6 to ES6.

Traceur

Google maintains a project called "Traceur" [1], which is exactly tasked with transpiling ES6 features into pre-ES6 (mostly ES5, but not all!) for general usage. The TC39 committee relies on this tool (and others) to test out the semantics of the features they specify.

What does Traceur produce from our snippet? You guessed it!

```
{
        try {
                throw undefined;
        } catch (a) {
                a = 2;
                console.log( a );
        }
}

console.log( a );
```

So, with the use of such tools, we can start taking advantage of block scope regardless of if we are targeting ES6 or not, because `try/catch` has been around (and worked this way) from ES3 days.

Implicit vs. Explicit Blocks

In Chapter 3, we identified some potential pitfalls to code maintainability/refactorability when we introduce block-scoping. Is there another way to take advantage of block scope but to reduce this downside?

Consider this alternate form of `let`, called the "let block" or "let statement" (contrasted with "let declarations" from before).

```
let (a = 2) {
        console.log( a ); // 2
}

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

Instead of implicitly hijacking an existing block, the let-statement creates an explicit block for its scope binding. Not only does the explicit block stand out more, and perhaps fare more robustly in code refactoring, it produces somewhat cleaner code by, grammatically, forcing all the declarations to the top of the block. This makes it easier to look at any block and know what's scoped to it and not.

As a pattern, it mirrors the approach many people take in function-scoping when they manually move/hoist all their `var` declarations to the top of the function. The let-

statement puts them there at the top of the block by intent, and if you don't use `let` declarations strewn throughout, your block-scoping declarations are somewhat easier to identify and maintain.
But, there's a problem. The let-statement form is not included in ES6. Neither does the official Traceur compiler accept that form of code.

We have two options. We can format using ES6-valid syntax and a little sprinkle of code discipline:

```
/*let*/ { let a = 2;
        console.log( a );
}

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

But, tools are meant to solve our problems. So the other option is to write explicit let statement blocks, and let a tool convert them to valid, working code.

So, I built a tool called "let-er" [^note-let_er] to address just this issue. *let-er* is a build-step code transpiler, but its only task is to find let-statement forms and transpile them. It will leave alone any of the rest of your code, including any let-declarations. You can safely use *let-er* as the first ES6 transpiler step, and then pass your code through something like Traceur if necessary.

Moreover, *let-er* has a configuration flag `--es6`, which when turned on (off by default), changes the kind of code produced. Instead of the `try/catch` ES3 polyfill hack, *let-er* would take our snippet and produce the fully ES6-compliant, non-hacky:

```
{
        let a = 2;
        console.log( a );
}

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

So, you can start using *let-er* right away, and target all pre-ES6 environments, and when you only care about ES6, you can add the flag and instantly target only ES6.

And most importantly, **you can use the more preferable and more explicit let-statement form** even though it is not an official part of any ES version (yet).

Performance

Let me add one last quick note on the performance of `try/catch`, and/or to address the question, "why not just use an IIFE to create the scope?"

Firstly, the performance of `try/catch` *is* slower, but there's no reasonable assumption that it *has* to be that way, or even that it *always will be* that way. Since the official TC39-approved ES6 transpiler uses `try/catch`, the Traceur team has asked Chrome to improve the performance of `try/catch`, and they are obviously motivated to do so.
Secondly, IIFE is not a fair apples-to-apples comparison with `try/catch`, because a function wrapped around any arbitrary code changes the meaning, inside of that code, of `this`, `return`, `break`, and `continue`. IIFE is not a suitable general substitute. It could only be used manually in certain cases.
The question really becomes: do you want block-scoping, or not. If you do, these tools provide you that option. If not, keep using `var` and go on about your coding!
[^note-let_er]: let-er