Lesson 3.2 - Block Scoped Variables

Let's explore block scoped variables shall we? Block scoped variables are simply variables that are local to the block in which they were defined. Let's explore this a bit with some code samples.

The Basics

If we were to use `var` to declare a variable in this snippet, everything would work just fine:

```
for (var i = 0; i < 3; i++) {
  var j = i * i
}

console.log(j) // This will print '4', and is fine
```

But let's say we use `let` to declare that variable:

```
for (let i = 0; i < 3; i++) {
  let j = i * i
}

console.log(j) // This will throw, because let is local to the `for` loop.
```

This is a subtle difference, but an important one. This really helps to keep variables scoped to their local block, and avoids confusion for other developers (and yourself).

Nested Scopes

Scoping also works in situations where the blocks are nested:

```
function foo() {
  let bar = 'Hello World!';

  return function() {
    return bar;
  }
}
```

In this example, the returned function has access to the bar variable because it was declared in the outer scope. The function that is returned from the foo function has access to all variables declared in the scope of the foo function.

However, this will throw an error:

```
function foo() {
  for (let i = 0; i < 3; i++) {
    let j = i;
  }

  return j; // This is not allowed, as j was declared inside a different scope
}
```

Taking this one step farther, we can do something like this:

```
let j = 0;

function foo() {
  for (let i = 0; i < 3; i++) {
    j += 1;
  }
}

foo()
console.log(j)
-> 6
```

This isn’t very good programming practice, but it clearly demonstrates how nested scoping works. The j variable is available in all child scopes.

Scoping with switch statements

Scoping also applies to switch statements. Adding curly braces to your case statements will encapsulate them in their own scope.

```
const i = 0;

switch (i) {
  case 0:
    const label = 'zero';
    console.log(label);
    break;
  case 1:
    const label = 'one';
    console.log(label);
    break;
}
```

This will throw an error because you are attempting to define the `label` const twice inside of the same scope. However, if we rewrite that like this:

```
const i = 0;

switch (i) {
  case 0: {
    const label = 'zero';
    console.log(label);
    break;
  }
  case 1: {
    const label = 'one';
    console.log(label);
    break;
  }
}
```

This is perfectly acceptable, because we have created an isolated scope for each case statement.

Wrapping up

Anything inside curly braces ({}) is considered to be inside of a block, and
you can create a block all by itself:

```
{
  // This is inside of an isolated block scope
  let foo = 1;
}

console.log(foo); // `foo` is undefined in this scope
```

Constants behave in exactly the same way as `let` when it comes to scoping. It is for this reason that it is preferred to use `const` and `let` over `var` as often as possible.