Scope in JavaScript

I've mentioned the term **scope** a couple of times in this course and I haven't really gone into details about defining it but I'd like to show you more about how scope work in JavaScript and why you need to keep it in mind.

Functions with Var Statements

You might have noticed when we've been making functions, we've been creating variables inside those functions to declare local variables such as counters or things that'll be in the return value. Let's go back to one of the examples that we used in the last set of exercises.

```
var greeting = "Hello";
var items = ["World"];

function greeter(str, arr) {
  var counter; // local variable in the function
  for (counter = 0; counter < arr.length; counter++) {
    console.log(str + " " + arr[counter]);
  }
}

greeter(greeting, items);
// "Hello World"

console.log(counter);
// "error" (ReferenceError: counter is not defined)
```

If we were to try to `console.log` the `counter` variable outside of the function `greeter`, it will be `undefined`. `counter` does not exist outside the scope of the function `greeter`.

In JavaScript, you create a new scope every time that you create a new function. And scope is defined such that variables created within the scope of a function using the `var` keyword only exist within that function, and can't be accessed outside of that function.

Var Statements Limit Scope

The reason that works is because we're using the `var` statement to create our variables. The `var` statement limits the scope of the variable to wherever to it was declared. A variable declared without a `var` statement is valid in JavaScript but it will be applied to the global scope which means that it will exist outside the scope of your script and be available inside of the script as well but that can cause problems.

I'm going to show you how global variables work and then I'm going to tell you never to use them but you need to understand what happens with global variables. We're going to create a quick function and it's just going to be called `example`.

```
function example() {
  var local = "Only available inside example()"; // with var
  global = "Available anywhere once executed"; // without var
}

console.log(local);
// "error" (ReferenceError: local is not defined)
```

```
console.log(global);
// "error" (ReferenceError: local is not defined)

example(); // execute the example function

console.log(local);
// "error" (ReferenceError: local is not defined)

console.log(global);
// "Also available anywhere"
```

The problem with that is that we don't know at any point in the program whether `example` has been executed. `global` could be defined or not defined and we have no way of understand that, we don't have a modular approach to how this function works, if it's plying and creating global variables. This is one of the many reasons that coding environments that checkthe validity of your job JavaScript code while you're coding will tell you that variables created without the `var` keyword should not be used.Variables can be inherited by functions from their parent scope.

Variable Scope is Inherited One Way

Variables can be inherited by functions from their parent scope and functions inherit all of the variables that existin the scope in which they were created which makes sense, because a function needs access to the variables that were available at the time that it was created.

```
var phrase = "always available";

function example() {
  var local = "Only available inside example()";
  console.log(phrase);
  console.log(local);
}

example();
// "always available"
// "Only available inside example()"

console.log(phrase);
// "always available"

console.log(local);
// "error" (ReferenceError: local is not defined)
```

Understanding the difference between a local variable and a variable that exists outside of the scope of a function will help you avoid undefined variable errors like that.

Declaring Variables First

You might also have noticed that inside of each function fact inside each of each program that we've been writing, we've been declaring all of our variables at the very top of the code that we're writing. JavaScript will hoist the variable declarations to the top of whatever scope that that variable exists in. For that reason, even though JavaScript will let you declare variables anywhere in the body of your code, it's a very good idea to start each function or script by declaring any variables you plan to use within that scope.

Any time you come across a variable you need to define, declare it at the top of whatever scope it's going to live in. It will help you keep track of where your variables live. Understanding variable hoisting can help you some of the trickier problems to track down in JavaScript.

```
// one script that uses a counter
count = 5;

// another script that sets count based on a price
function thousands(price) {
  count = price * 1000;
  return(count);
}
thousands(50);

// another script that uses count for an iterator
```

```
for (count = 0; count < 6; count++) {
  // do something six times
}

// A whole bunch of code later...

console.log(count);
// could be anything
```

Variable hoisting comes about because when JavaScript processes a script to run it, the compiler does two passes. First it goes through all of the code, defines all of the functions, and declares all of the variables. Then on the second pass, it goes through and runs the statements in order. Functions are declared and variables are declared and hoisted to the top of their scope during the first pass before any execution happens.

The Global Scope

So when considering what scope a variable is going to live in, JavaScript first looks to see what it's container is. If it's not contained within a function, that variable is assigned to the global scope. All scripts that are being executed at the same time share space in a single global object. That global object is called window in browsers, and it's called global on the server, for example, in Node. Because of this,you need to be careful about assigning variables to this global scope.

```
var count = 5;

console.log(count);
// 5
console.log(window.count);
// 5

window.count = 10;
console.log(count);
// 10
console.log(window.count);
// 10
```

Every script that runs simultaneously in your browseris going to have access to the `window` object, just like every script that runs simultaneously on your server is going to have access to the global object.

What's Wrong with Global Scope?

So, what's wrong with assigning variables to the global scope? The problem is that different parts of your code may reuse a common variable name. And no matter how careful you think you're being, this can happen even within the smallest and most ordinary of scripts. Imagine how much more complicated that could be if you're simultaneously running multiple scripts from multiple programmers, all of them loaded on the same page at the same time.

```
function predefined() {
  var before = 5; // declared and initialized
  var after = 10; // declared and initialized
  return(before + after);
}

function undefinedAfter() {
  var before = 5; // declared and initialized
  return(before + after);
  var after = 10; // initialized, declaration hoisted
}

function undeclaredAfter() {
  var before = 5; // declared and initialized
  return(before + after);
}

console.log(predefined());
// 15
```

```
console.log(undefinedAfter());
// NaN

console.log(undeclaredAfter());
// error: ReferenceError: after is not defined
```

The question is, after doing all of those things, what's the value of `count` and the point is that we don't really have any way of knowing. Depending on what happened in that bunch of other code and what order these things were declared in, because we use the same count variable over and over again, all in the same scope, we've lost track of what the value of `count` actually is going to be.

This gets even worse if we leave off the `var` keyword on `count`. No matter how carefully we've managed our scripts, the variable `count` could exist anywhere in any of the scripts that we're running.

Basically, you want to be careful not to set variables on the global scope and you want to pay attention to warnings that tell you not to use global scope.

Ways to Isolate Variable Scope

There are some ways to isolate variable scope and these are good practices to keep in mind when you're coding. Keep local variables out of the global object by declaring them inside of functions, wrap all of your code inside of an immediately invoked function expression or write your code inside of custom objects that contain all of the methods you're going to be using and encapsulate them that way.

I'll give you an example of each of these three ways to encapsulate your variable scope.

```
// local to a function
function thousands(price) {
  var count = price * 1000;
}

// immediately invoked function expressions
(function() { // start a function expression
  var count;
  for (count = 0; count < 6; count++) {
    // do something six times
  }
}()); // execute that function expression

// local to a private object (less safe)
var myApp = {};
myApp.count = "Hello".length;

console.log(count);
// error: ReferenceError: count is not defined
```

The first way we've been talking about when you declare a function, declare your variables inside of that function using the `var` keyword. This way you can be confident that those variables will only exist within the scope of that function.

A second way is by creating an immediately invoked function expression. So we execute the function as soon as it's declared and we're going to do to that the same way we execute any function with a set a parentheses and we end with a semicolon.

A third way is to start by declaring an object that's going to contain all of the code for your application.

Being a good developer citizen while you're programming in JavaScript, means understanding how you isolate your variables from the global scope, so that your variables don't leak into other people's programs and vice versa.