**Closures and Scope**

**Scope**

The **scope** of a method is the set of variables that are available for use within the method. The scope of a function includes: 1. the function's arguments; 2. any local variables declared inside the function; 3. **any variables that were already declared when the function was defined**.

Consider this example:

function sayHelloNTimes(name, n) {

function greet() {

console.log( `Hi, ${name}!`);

}

for (let i = 0; i < n; i++) {

greet();

}

}

sayHelloNTimes("Bob", 3); // logs 'Hi, Bob!' x3

sayHelloNTimes("Sally", 6); // logs 'Hi, Sally!' x6

In the example above, the variable name is referenced by greet, even though it was never declared within greet. This is possible because **a nested function's scope includes variables declared in the scope where the function was nested.**

**Closures**

Functions such as greet that use (ie. **capture**) such variables (ie. **free variables**) are called **closures**.

**Free variables can be modified** by closures. Consider this function:

function sum(nums) {

let count = 0;

function addNum(num) {

count += num;

}

for (let i = 0; i < nums.length; i++){

addNum(nums[i]);

}

return count;

}

sum([1, 3, 5]) // => 9

**Applications**

**Passing Arguments Implicitly**

We can use closures to pass down arguments to helper functions without explicitly listing them as arguments.

function isPalindrome(string) {

function reverse() {

return string.split('').reverse().join('');

}

return string === reverse();

}

**Private State**

Another major use of closures is to create private states. For example:

function createCounter() {

let count = 0;

// ++count increments the value of the count variable by 1 before it is evaluated in the statement.

// count++ returns the value of the count variable before it is incremented by 1. If we used count++ instead, our counter would be off by 1.

return () => ++count;

}

let counter = createCounter();

console.log(counter()); // => 1

console.log(counter()); // => 2

counter.count; // undefined

let counter2 = createCounter();

console.log(counter2()); // => 1

[More info about ++count here](https://stackoverflow.com/questions/3469885/somevariable-vs-somevariable-in-javascript)

By **closing over** (or **capturing**) the count variable, each function that is returned from createCounter has a private, mutable state that cannot be accessed externally.

Compare that implementation against this one:

function Counter () {

this.\_count = 0;

}

Counter.prototype.fire = function () {

this.\_count += 1;

return this.\_count;

}

let counter = new Counter();

counter.fire(); // 1

counter.fire(); // 2

counter.\_count // 2

counter.\_count = 0 // 0 (this works);

One advantage of the closure way is that the count is **truly private**. In the first example, there is no way any method beside the closure itself can access the count state. In the second example, a foolish user might set counter.\_count inadvertently.

**Global Scope**

JavaScript has global scope, represented by the 'window' object in the browser and the 'global' object in Node.js. Adding attributes to these objects makes them available throughout a program.

function theBest() {

window.realMVP = 'you';

}

theBest(); // initializes realMVP on the global scope

window.realMVP; // 'you'

function whoDaBest() {

return realMVP; // 'you'

}

whoDaBest(); // 'you'

While useful on occasion, global variables are usually best avoided, as they give too much code access to their values, increasing the likelihood of bugs.

"use strict";

A common mistake new JS developers commit is to unintentionally create global variables. This happens if you declare a variable without the var, let, or const keywords anywhere in your code, and can lead to strange behavior. Consider:

window.local; // undefined

function subroutine(){

local = 'oops';

}

subroutine();

window.local // 'oops';

Thankfully, modern JS runtimes support *strict mode*, which prohibits variable declaration without var, let, or const.

"use strict";

window.local; // undefined

function subRoutine(){

local = 'oops';

}

subRoutine(); // ReferenceError: 'local' is not defined

**Note**: "use strict" does not work in the Node CLI or the Dev Tools console.

**References**

**Read these!**

* [JavaScript Closures with Ease](http://javascriptissexy.com/understand-javascript-closures-with-ease/)
* [SO: How Do JS Closures Work](http://stackoverflow.com/questions/111102/how-do-javascript-closures-work) (the first answer)