JavaScript Closures

COEN 161

Local Variables

- All variables inside a function are local to that function and can be accessed inside that function

```
function myFunction() {

    var a = 4;

    return a * a;

}
```

Global Variables

- Functions can also access any variables defined outside the functions, such as global variables

```
var a = 4;

function myFunction() {

    return a * a;

}
```

Variable Lifetime

- Global variables are attached to the window object and are available as long as your browser window is open
- Local variables are only available within the function when it is invoked, and disappear when the function finishes

Example - Counter

- If you wanted to build a counter function, you can use a global variable like this

```
var counter = 0;

function add() {
    counter += 1;
}

add();
add();
add();

// the counter is now equal to 3
```

Example - Counter

- This works, but the main problem is that since the counter is a global, any other script in the page can change the counter without calling add()!
- We can fix this by making the counter a local variable

```
function add() {
    var counter = 0;
    counter += 1;
}

add();
add();
add();

// the counter should now be 3, but it does not work !
```

Example - Counter

- The problem is that the counter is declared each time we invoke add, therefore it always starts life as 0 and gets incremented to 1

```
function add() {
    var counter = 0;
    counter += 1;
}
```

Nested Functions

- We know all functions have access to the global scope
- But functions also have access to the variables in the function “above” them

```
function add() {

    var counter = 0;

    function plus() {counter += 1;}

    plus();

    return counter;

}
```

Example - Counter

- This almost solves our counter problem but we need a way to access the plus function from the outside and only set the counter once, we need a **closure**

```
function add() {

    var counter = 0;

    function plus() {counter += 1;}

    plus();

    return counter;

}
```

Closures

- The following pattern is called a *self-invoking function*

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();

add();
add();
add();

// the counter is now 3
```

Closures

- The variable add holds the function that was returned by the function expression in between the parentheses `(function(...){...})();`
- The counter is declared inside the self-invoking function and set to 0, this only happens once, when the function invokes itself
- The function that is returned has access to the counter because the counter is declared *outside* of it, this is called a ***closure***

“A closure is a function having access to the parent scope, even after the parent function has closed.”

Resources

https://www.w3schools.com/js/js_function_closures.asp