The Learning House

427 S. 4th Street #300

Louisville KY 40202

Function Expressions

.NET Cohort

Coding Bootcamp

SOFTWARE GUILD

Declaration vs Function Expressions

As a reminder, function declaration is much like a C# function where you declare a method name and arguments.

Function declarations are read in before any code executes, so the order in the file does not matter.

Function expressions are assigned to variables or declared in-line as responses to events. These functions are *anonymous* and thus have no name.

You must assign a function expression before it is used.

```
// declared functions are read before the code executes
// this code will execute just fine
sayHi();
function sayHi() {
    alert('Hi!');
}
```

```
// this is a function expression
// sayHi will error out
sayHi();
var sayHi = function() {
    alert("Hi!");
};
```

Assigning Function Expressions Grants Flexibility

Although JavaScript doesn't support interfaces the way other languages do, we can assign function expressions conditionally and pass them around in variables.

```
var sayHi;

if (condition) {
    sayHi = function () {
        alert("Hi!");
    };
} else {
    sayHi = function () {
        alert("Yo!");
    };
}
```

Recursion

A *recursive function* is formed when a function calls itself by name. Most languages support recursion in some form.

```
function factorial(num){
    if (num <= 1){
        return 1;
    } else {
        return num * factorial(num-1);
    }
}
```

Closures

Closures are functions that have access to variables from another function's scope. Usually this happens by creating a function in a function.

```
function createComparisonFunction(propertyName) {
    // this can see propertyName since it is in scope
    return function (object1, object2) {
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];
        if (value1 < value2) {
            return -1;
        } else if (value1 > value2) {
            return 1;
        } else {
            return 0;
        }
    };
}
```

this Object and Closures

The *this* object is bound at runtime based on the context in which the function is executed.

So in a global function, this is equal to window (nonstrict) or undefined (strict). However this is equal to the object itself when called as an object method.

Each function automatically gets two special variables: *this* and *arguments* which an inner function can not directly access from an outer function.

Since getNameFunc returns a closure func, it can't access the this that points to the object, so it will default to window (in nonstrict mode)

```
var name = 'The Window';
var object = {
    name: 'My Object',

    getNameFunc : function(){
        return function(){
            return this.name;
        };
    }
};

alert(object.getNameFunc()()); // The Window
```

But what if…

What if you really want to access the object from the closure? It's actually simply assigning this to a variable so you can pass it down like so:

```
var name = 'The Window';

var object = {
    name : 'My Object',

    getNameFunc: function () {
        var that = this;

        return function () {
            return that.name;
        };
    }
};

alert(object.getNameFunc()()); //"My Object"
```

Mimicking Block Scope

As we talked about previously, JavaScript has no concept of block scoping, so any variables defined in a block of statements are created in the containing function, like the top example.

Function expressions can fix this. The syntax is a little odd to someone coming from C#, but this causes the alert(i) to throw an error as we would expect.

Plugin authors often do this so they don't pollute the global scope with their variables and accidently cause collisions.

```
function outputNumbers(count) {
    for (var i = 0; i < count; i++) {
        alert(i);
    }

    alert(i); //count
}
```

```
function outputNumbers(count) {
    (function () {
        for (var i = 0; i < count; i++) {
            alert(i);
        }
    })();

    alert(i); //causes an error
}
```

Using this to mimic private and public members

We can use the this keyword inside functions to define public functions for accessing members that would normally be private.

Using a function constructor object allows for all private members to be passed in as parameters or declared in the body, while any function members with the this() keyword will be public but also allow visibility to the private members.

```
function Person(name) {
    this.getName = function () {
        return name;
    };
    this.setName = function (value) {
        name = value;
    };
}

var person = new Person('Eric');
alert(person.getName()); // Eric
person.setName('Jennie');
alert(person.getName()); //Jennie
```

Higher Order Functions

Higher-order functions are basically functions that accept another function as arguments.

In the example to the right, the function (sometimes referred to as the *callback*) is passed to the filter method which will execute the function on each member of the array.

The callback must return a true value for the item to be returned (this is similar to LINQ filters).

.filter() is a useful built-in JavaScript higher order function.

```
var pets = [
  { name: 'Woot', type: 'dog', age: 8 },
  { name: 'Angry Cat', type: 'cat', age: 14 },
  { name: 'Dash', type: 'dog', age: 7 },
  { name: 'Pi', type: 'parrot', age: 2 }
];


var dogs = pets.filter(function (pet) {
    return pet.type === 'dog';
});
```

.map()

Map is another useful higher order function. Whereas filter returns a boolean to determine whether an element should be included in the result, map accepts a callback function that is called for every item in the array.

You can even chain these together like so ->

```
var pets = [
  { name: 'Woot', type: 'dog', age: 8 },
  { name: 'Angry Cat', type: 'cat', age: 14 },
  { name: 'Dash', type: 'dog', age: 7 },
  { name: 'Pi', type: 'parrot', age: 2 }
];


var dogs = pets
    .filter(function (pet) {
        return pet.type === 'dog';
    })
    .map(function(pet) {
        return pet.name;
});

// dogs will contain the array ['Woot', 'Dash']
```

.reduce()

Reduce() can be a bit scary to newcomers, but it's really not all that complicated.

Reduce iterates over all the values of an array and passes both the value returned from the prior run and the current value.

A simple example of this in action is to the right where we do a running total on the ages of our pets. The previous value is the running total and the current value is the age we want to add to it.

Reduce is often used with map, so you'll hear developers talking about "map reduce". This is what they mean.

```
var pets = [
  { name: 'Woot', type: 'dog', age: 8 },
  { name: 'Angry Cat', type: 'cat', age: 14 },
  { name: 'Dash', type: 'dog', age: 7 },
  { name: 'Pi', type: 'parrot', age: 2 }
];


var totalDogYears = pets
    .filter(function (pet) {
        return pet.type === 'dog';
    })
    .map(function(pet) {
        return pet.age;
    })
    .reduce(function(runningTotal, age) {
        return (runningTotal || 0) + age;
});

// totalDogYears will be 15
```

Coming Soon – Arrow Functions

The next JavaScript standard, ES6, will include arrow functions (we call them lambdas). So you will be able to write the previous query like so:

```
var totalDogYears = pets
  .filter((pet) => pet.type === 'dog')
  .map((pet) => pet.age)
  .reduce((runningTotal, age) => (runningTotal || 0) + age);
```

Conclusion

Function expressions are very useful tools when working with JavaScript, especially when writing APIs and SPA applications.

Always keep in mind the behavior of functions within functions (closures) and their impact on the this keyword.

If you happen to need block scoping or private variables closures are the way to go about it!

Just watch out for memory leaks, since closures stay in memory until they no longer exist.

Master filter, map, and reduce if you want to be a serious JavaScript developer.