Web Advanced: Javascript APIs

“We will learn JavaScript properly. Then, we will learn useful design patterns. Then we will pick up useful tools for making cool things better.”

FALL 2018

SESSION #9

FUNCTIONAL PROGRAMMING

WRITING MODULES

jaink@newschool.edu

https://canvas.newschool.edu/courses/1407281

https://classroom.github.com/classrooms/42809645-parsons-web-advanced-javascript-fall-2018

RECAP

WHAT IS FUNCTIONAL PROGRAMMING

- FP vs OOP
- Javascript Advantage
- Programming Concepts
  - Pure Functions
  - No side-effects
  - Return Transparency
  - Higher-order Functions

THE BASICS

- In Javascript, functions are objects, and they behave like other variables/values with their own properties and methods and can be passed as arguments and returned by functions.
- This makes functions very flexible and they can be used to create modules and libraries for cleaner code and speedier execution.
- Functional programming uses functions and their features to build out the operations to perform single tasks through encapsulation. Then it can combine them all to build out more complex structures.
- Each function performs a task and returns a value that can be worked on by another (doesn't change or use the outside world).
- The emphasis is on simplicity where data and functions are concerned.

RETURNING FUNCTIONS

Returns a function based on specific params:

```
function power(x) {
  return function(power) {
    return Math.pow(x,power);
  }
}
let twoExp = power(2);
twoExp(5);
power(2)(5);
```

CALLBACKS

Functions that are sent as arguments to another function. Allows for more generalized function definitions:

```
// pass function as a parameter
const convertToCelsius =  function(deg_fah) {
     let converted_deg = (deg_fah-32) * 5/9;
     return converted_deg;
};
const convertToFahrenheit =  function(deg_cent) {
     let converted_deg = (deg_cent * 9/5) + 32;
     return converted_deg;
};
// pass function as a parameter
function convert(converter, temperature) {
     console.log(converter(temperature));
}
convert(convertToCelsius, 23);
convert(convertToFahrenheit, 23);
```

CLOSURES

Function that contain functions that get access to the outer function's variables and params, and stay alive after the initial call:

```
function counter(start){
  i = start;
  return function() {
    return i++;
  }
}
let count = counter(1); // start a counter at 1
count();

function multiplier(factor) {
  let ret = function(number) {
    return number * factor;
  };
  return ret;
}
const square = multiplier(2);
const cubed = multiplier(3);

square(3);
cubed(3);
multiplier(2)(3); // same as above square(3)
```

RECURSIVE FUNCTIONS

Calls itself until a certain condition is met - useful for iterative processes:

```
function factorial(n) {
  if (n === 0) {
    return 1
  } else {
    return n * factorial(n - 1);
  }
}

factorial(10);
```

CURRYING

Transforms a function into another with lesser number of arguments. It can help avoid continuously passing the same variables:

```
function multiplier(x,y) {
  if (y === undefined) {
    return function(z) {
   return x * z;
    }
  } else {
    return x * y;
  }
}

multiplier(3,5); //this works normally
quadrupler = multiplier(4);
quadrupler(5);
```

PROMISES

Created to better manage the results of asynchronous operations. Cleans up Callback Hell:

```
const promise = new Promise( (resolve, reject) => {
    // initialization code goes here
    if (success) {
        resolve(value);
    } else {
        reject(error);
    }
});

const dice = {
     sides: 6,
     roll: function() {
          return Math.floor(this.sides * Math.random()) + 1;
     }
}

const promise = new Promise( (resolve,reject) => {
     const n = dice.roll();
     (n > 3) ? resolve(n) : reject(n);
});

promise.then( result => console.log(`Yes! I rolled a ${result}`),
              result => console.log(`Drat! ... I rolled a
          ${result}`)
          );

promise.catch( result => console.log(`Damn! ... I rolled a
${result}`));
```

IIFEs

An Immediately Invoked Function Expression is a function that gets invoked as soon as it’s defined.

- ➔ **Keeps scope of variables local and doesn't clutter global.**
- ➔ **Also called Anonymous Closures**
- ➔ **One of the most powerful Javascript features!**

```
(function(){
  var temp = "world";
  console.log("Hello " + temp);
}());
```

IIEFs - block scopes

Most languages, let a variable have scope inside a code, but not JavaScript (pre ES6) - variables live outside a block. IIFE can mimic the block scope:

```
var list = [1,2,3];
for (var i = 0, max = list.length ; i < max ; i++
){
  console.log(list[i]);
}
console.log(i); //i is still outside


var list = [1,2,3];
(function(){
    for (var i = 0, max = list.length ; i < max ;
i++ ){
        console.log(list[i]);
    }
}());
console.log(i); // i is not available outside the
for block
```

IIEFs - global import, local scope

IIFE can access the global scope just like any function. But it's better practice to pass a global variable to it to keep the block self-contained:

```
var list = [1,2,3];
(function(my_list){
    for (var i = 0, max = my_list.length ; i < max ;
i++ ){
        console.log(my_list[i]);
    }
}(list));
console.log(i); // i is not available outside the for
block

// this is how we use jQuery!
(function($){
  // we have access to globals jQuery (as $)
  // bind click event to all internal page anchors
  $(".menu-link").bind("click touchstart", function (e)
{
      // prevent default action and bubbling
      e.preventDefault();
      e.stopPropagation();
      $("#off-canvas-nav").toggleClass("active");
      $("#wrapper").toggleClass("active");
  });
})(jQuery);
```

IIEFs - with ES6

Using ES6 variable initializers let, const, you don't need IIFEs since these 2 are block scoped:

```
{
    const list = [1,2,3];
    for (let i = 0, max = list.length ; i < max ; i++
){
        console.log(list[i]);
    }
};
console.log(i); // i is not available outside the for
block
```

MODULES

➔ The Module Pattern

THE MODULE PATTERN

- ➔ Self-contained piece of code that provides functions and methods that can then be used elsewhere
- ➔ Improves usability, readability, maintainability, namespacing, error-checking
- ➔ Generally organized by distinct functionality
- ➔ Modules store private/public methods and properties inside an object that can interact with the outside through an API (similar to Classes)

1. ANONYMOUS CLOSURE

An Anonymous function has its own evaluation environment which is immediately evaluated, hiding variables from the parent (global) namespace.

```
(function() {
  "use strict";

  function square(x) {
    return x * x;
  }
  function sum(array, callback) {
    if (typeof callback === "function") {
      array = array.map(callback);
    }
    return array.reduce(function(a,b) { return a + b;
});
  }
  function mean(array) {
    return sum(array) / array.length;
  }
  function sd(array) {
    return sum(array,square) / array.length -
square(mean(array));
  }
  console.log(sd([1,3,5,7]));
}());
```

2. GLOBAL IMPORT

An Anonymous function which receives globals as parameters.

```
(function(my_list) {
  "use strict";
  console.log(my_list);
  function square(x) {
     return x * x;
  }
  function sum(array, callback) {
    if (typeof callback === "function") {
      array = array.map(callback);
    }
    return array.reduce(function(a,b) { return a + b;
});
  }
  function mean(array) {
    return sum(array) / array.length;
  }
  function sd(array) {
    return sum(array,square) / array.length -
square(mean(array));
  }
  console.log(sd(my_list));
}([1,3,5,7]));
```

3. REVEALING MODULE PATTERN

uses an IIFE to return the module as an object that’s stored in a global variable, keeping all methods and variables private until explicitly exposed:

```
var Stats = (function(my_list) {
  "use strict";
  function square(x) {
    return x * x;
  }
  function sum(array, callback) {
    if (typeof callback === "function") {
      array = array.map(callback);
    }
    return array.reduce(function(a,b) { return a + b;
});
  }
  function mean(array) {
    return sum(array) / array.length;
  }
  function sd(array) {
    return sum(array,square) / array.length -
square(mean(array));
  }
  return {
    mean: mean,
    standardDeviation: sd
  };
}(counts));
var counts = [1,3,5,7];
Stats.mean(counts);
Stats.standardDeviation(counts);
```

ES6 MODULES

ES6 includes a native format for importing and exporting modules from different files

- ➔ import function brings the module into the current namespace.
- ➔ export exposes the elements in the module to public so it can be used.

```
//------ scripts.js ------
import {stats} from './stats.js';

console.log(stats().mean(counts));
console.log(stats().standardDeviation(counts));
```

```
//------ stats.js ------
const Stats = function(my_list) {
     ... ...
}
export const stats = Stats;
```

ES6 MODULES

ES6 includes a native format for importing and exporting modules from different files

- ➔ import function brings the module into the current namespace.
- ➔ export exposes the elements in the module to public so it can be used.

```
//------ scripts.js ------
import {stats} from './stats.js';

console.log(stats().mean(counts));
console.log(stats().standardDeviation(counts));
```

```
//------ stats.js ------
const Stats = function(my_list) {
     ... ...
}
export const stats = Stats;
```

EXAMPLES

Assignment

Next Steps

➔ Introduction to Modern Development Workflows