CS 4220

- Current Trends in Web Development -

Application Design and Development with Node.js

Prof. Cydney Auman

AGENDA

01 First Class Functions

02 Higher Order Array Methods

03 ES6 Classes

04 Module Design Pattern

Review

We continued to understand the meaning of mutation in JavaScript. We discussed Objects with code demo's showing they are mutable.

We also looked at Object methods and various ways to approach Object iteration.

We added more ES6 concepts which were both applicable to Objects and Arrays. We looked at destructuring, spread and rest syntax via slides and code demo.

First Class Functions

First Class functions in a programming language are functions that are treated like any other variable. JavaScript supports First Class functions. A function can be assigned as a value to a variable, passed as an argument to other functions, and can even be returned by a function. And JavaScript we can take this a step further and store functions inside objects.

One of the benefits with using **First Class** functions is that when coding we have the power to abstract the processing of our data, we reduce complexity by removing the unnecessary. First Class functions allow us to process data in a eloquent and succinct way.

Higher-Order Functions

A type of First Class Function that that operates on other functions, either by taking them as arguments or by returning them, are called **higher-order functions**.

```
const repeat = (n, action) => {
    for (let i = 0; i < n; i++) {
        action(i);
    }
};
repeat(5, console.log);
```

Array Method: forEach

The `forEach()` method executes the provided function once for each array element. The `forEach()` method *does not* return. Instead the `forEach()` method is an alternative to the standard for loop.

Technically, the for loop is more performant - benchmarking shows on a million entries for loop performs approx 3x faster.

```
const inventory = [1, 5, 7, 3, 2];
inventory.forEach((count) => {
    console.log(`Unit: ${count}`)
})
```

Array Method: map

The `map()` method returns a new array that is populated with the results of calling a provided function on every element in the original array. The returned array will always be the same size as the input array.

```
const squares = [2, 3, 4, 5];
const results = squares.map(n => {
   return n * n;
});
console.log(results);
// [ 4, 9, 16, 25 ]
```

Array Method: filter

The `filter()` method returns a new array with all elements that pass the expression implemented in the provided function. The new array length will vary as only elements the pass the expression will be returned in that array.

```
const mixtypes = [5, 2, 'a', 4, true, 'b', 'c', 7, false, 8];
const numArray = mixtypes.filter(element => {
    if (typeof element === 'number') {
        return element;
    }
});
console.log(numArray);
// [ 5, 2, 4, 7, 8 ]
```

Array Method: sort

The `sort()` method sorts the elements of an array in place and returns the sorted array. This method optionally accepts a compare function to perform custom sorting. The compare function that should be used is dependent on the type of comparison being performed.

```
function compareString(a, b) {
   if (a is less than b by some ordering criterion) {
       return -1;
   }
   if (a is greater than b by the ordering criterion)
       return 1;
   }
   return 0; // a and b are equal
}
```

```
function compareNumbers(a, b) {
   return a - b;
}
```

JavaScript Classes

Classes in JavaScript are ***not*** like classes in Java.

In languages like Java when you want a new object, you instantiate the class. This essentially tells the language to copy the methods and properties of the class into a new entity, called an **instance**. Once this instantiation happens it no longer has an active relation with the original class.

JavaScript *does not* have this concept of copying mechanics. Meaning that "instantiating" a class in JavaScript does create a new object, but *not* one that is independent of its original class. JavaScript creates an object that is *linked* to a prototype. ***Changes to that prototype propagate to the new object, even after instantiation.***

JavaScript Prototype

Every object in JavaScript has a built-in property, which is called its **prototype**. The prototype is itself an object and thus makes what is referred to as the prototype chain.

Because every object in JavaScript has this prototype which exists by default. This is why Object, Arrays, and even functions have properties and methods available in the prototype chain.

OK, but what about these properties and methods on primitive types?

Even though our primitive values are not objects - yes we still have access to their inherited properties. This is because the property accessor (dot) temporarily converts the primitive value to an object so that the method being called can be resolved up in **prototype chain**.

```
const csv = 'venus,earth,mars';
const arr = csv.split(',');
```

ES6 JavaScript Classes

JavaScript Classes introduced in ES6 are for the most part *syntactical sugar* over JavaScript's existing prototype-based inheritance. By *syntactical sugar* we mean that this class syntax makes it easier for us to read code without adding a lot of new functionality to ES5 style classes.

To declare a class, you use the **class** keyword. The optional **constructor** method is a special method for creating and initializing an object created with a class.

```
class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
const rectangle = new Polygon(10, 15);
```

rectangle - `linked to prototype →` Polygon - `linked to prototype →` Object

ES6 JavaScript Classes

JavaScript classes can only contain functions/method definitions. And unlike when creating object literals, you do not separate methods inside the class body with commas.

```
class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    getArea() {
        return this.height * this.width;
    }
}
const rectangle = new Polygon(10, 15);
rectangle.getArea();
```

ES6 vs ES5 JavaScript Classes

ES6

```
class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }

    getArea() {
        return this.height * this.width;
    }
}

const rectangle = new Polygon(10, 15);
const area = rectangle.getArea();
console.log(area);
```

ES5

```
function Polygon(height, width) {
    this.height = height;
    this.width = width;
}

Polygon.prototype.getArea = function () {
    return this.height * this.width;
};

var rectangle = new Polygon(10, 15);
var area = rectangle.getArea();
console.log(area);
```

JavaScript Class Inheritance

JavaScript *does not* offer multiple inheritance. There can be only be one prototype for an object. A class can then only extend from one other class.

However…there is a concept that help emulate multiple inheritance - this is know as **mixins**.

Wikipedia defines a **mixin** as a class containing methods that can be used by other classes without a need to inherit from it.

A straightforward way to implement a mixin in JavaScript is to create an object with the needed methods and then we can merge them into a prototype of any class. Instead - we use `Object.assign(Class, Object)`. There is no inheritance - all that is happening is method copying.

JavaScript Classes Mixin

```
class ComicCharacter {...}

class Hero extends ComicCharacter {...}

class Villian extends ComicCharacter {...}

const mixin = {
   iterateSidekicks() {
       for (let i = 0; i < this.sidekicks.length(); i++) {
       …
       }
   }
};

Object.assign(Hero.prototype, mixin);
```

JavaScript Module Design Pattern

The module design pattern was developed by a several people in 2003. However, it was later popularized by Douglas Crockford in his lectures. Today, this is one of the most common design patterns used in JavaScript.

As we can see - JavaScript has had modules for a long time. They were typically implemented using libraries, because ES6 was the first time that JavaScript introduced built-in modules.

The module pattern is used to keep some pieces of code independent of other components. This pattern provides both well-structured code and the concept of encapsulation.

There are several different variations and styles of the module design pattern since it has been around since 2003.

Module Design Pattern

We define a function. Just like a constructor this function can optionally accept arguments. Inside the body of the function we add logic by defining functions which perform related tasks. Finally, we return an object which contains the behavior we would like to expose publicly.

JavaScript does not have a keyword to indicate a variable is “private”. Thus developers have created a convention which is to prepend an underscore (_) to a variable name that should be considered “private”.

```
const polygon = (height, width) => {
    function getArea() {
        return height * width;
    }

    return {
        getArea
    };
};
const rectangle = polygon(10, 15);
const area = rectangle.getArea();
```

Classes vs Module Design Pattern

There is nothing negative or considered “bad practice” about using classes in JavaScript.

However, the problem that is typically seen is that this pattern is *misused* inside JavaScript codebases. Often time due to developers not fully understanding Prototype based classes.

Deciding between classes versus module design patterns - comes down to choosing the pattern that best addresses the issue/problem that needs to be solved in the language being used.

We should not select an OOP pattern of classes due to the fact we this is the only way we understand or like. Otherwise - selecting a design choice based on personal preference could make code harder for others to understand. Because this pattern is not a natural fit for the problem and/or programming language.

Resources

ES6 Features

- https://github.com/lukehoban/es6features

Modern JavaScript Tutorial

- https://javascript.info/class
- https://javascript.info/mixins

Mozilla Introduction to JavaScript

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules

Review and Prep

Review

- Review Slides
- Review and Run Code Examples

Homework 1 - DUE TODAY

- Due on Canvas - Wednesday 02/15 at 11:59pm PST
- Lab Time Today

Next Class

- Read Eloquent Javascript - Chapters 11
- Review various Homework 1 solutions

```
console.log('Week 04');
console.log('Code Examples');
```