ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

SESSION 10 - WHERE NOW?

1. closure
2. this
3. apply / bind
4. prototype
5. imports
6. js api
7. 3D

Reviews on:

o

8. organise code
9. tools for code organisation

1. closure

A closure is created when a `function` has access to its outer scope.

As soon as you create a function within an another function, you create a closure.

Example:

```
function makeFunc() {
  var name = "Mozilla";
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```

What is interesting about closures is that even when the outer function has returned, the inner function still has access to its outer scope (the scope defined by its wrapper function).

This means that a closure is formed of a function and its enviroment.

Practicly, closures can replace an object consisting of a single method.

They are extensively used in the module pattern.

2. `this`

When you create a function, the inner scope automatically gets a `this` keyword that is created and that gets assigned a value.

The value of the `this` keyword varies on how the function is called.

in global scope

In the global scope, outside of any function, `this` referes to the global object.

simple call

In this case, `this` defaults to the global scope (the global object).

```
function simpleCall () {
  return this
}
simpleCall();
```

function as an object method

When a function is defined as an object method, the `this` keyword takes the value of the object the method refers to.

```
var o = {
    property: "value",
    methodName: function () {
      return this
    }
}
o.methodName();
```

using a constructor

When a function is defined with a constructor, `this` is assigned the object created with the new operator.

```
function construct(){
  this.a = 37;
}

var o = new construct();
console.log(o.a); // logs 37
```

3. call and apply

When using a function's `call` or `apply` method, the `this` keyword is assigned a custom passed object.

```
function add(c, d){
  return this.a + this.b + c + d;
}

var o = {a:1, b:3};

add.call(o, 5, 7);

add.apply(o, [10, 20]);
```

Using the `this` keyword, you can build chainable methods!

All you need to do is for each function to return the current object it is working on, using `key`.

Build an object that has a "total" property (a number) and methods that add a number to this total, substract or increment this number by one or two.

You should then be able to do:

```
myObjectName.methodIncrement().methodAdd(10).methodSubstract(5); // etc.
```

```
var myObj = {
  total: 0,
  add: function (x) {
    this.total = this.total + x;
    return  this
  },
  sub: function (x) {
    this.total = this.total - x;
    return  this
  },
  increment: function () {
    this.total = this.total + 1;
    return this
  },
  decrement: function () {
    this.total = this.total - 1;
    return this
  },
  result: function () {
    console.log(this.total);
  }
}

myObj.add(10).decrement().result();
```

3. Call and Apply

Both are methods of `function`, they call a function, with the option to change the assigment of the `this` keyword.

Each method specifies waht to assign `this` as (the first argument as an object).

The rest of the arguments are specified normally in the case of `call`.

In the case of `apply`, the rest of the arguments are passed as an array.

4. prototype

Every instance in JavaScript is an object. This means that everything inherits properties and methods from the global object type.

Every type of object (array, function, strings, numbers etc.) will inherit the properties and methods of the type `object` and will add new ones to this.

This is the `prototype`.

You can add to the prototype of type by doing:

```
var myArray = [1, 3];

myArray.newMethod(); // error
```

```
Array.prototype.newMethod = function () {
  return this
}
```

```
myArray.newMethod(); // runs function defined above
```

Avoid adding properties and method to native objects and types!

You risk breaking existing functionalities and code.

You can however define your own object types, use as constructor to build other objects that will inherit the properties and methods of your constructors, and change the prototype of these.

You can check the `constructor` of an object by doing:

```
var myArray = [];

myArray.constructor // returns Array()

var myObject = {};

myObject.constructor // returns Object()

var myString = "";

myString.constructor // returns String()
```

Using the `new` operator, you can create a new instance of a constructor:

```
function Car(name) {
  this.name = name;
}

var myCar = new Car("Mustang");

typeof myCar // returns object
myCar.constructor // returns Car(name)
```

And then add to the prototype of the initial constructor function.

```
Car.prototype = {
  returnName: function(){return this.name;}
};
```

Using the previous exercise, can you build a constructor function for a calculator type of objects, that will have the same increment, decrement, add, substract methods?

```
function calculator () {
  this.total = 0;
  this.add = function (x) {
    this.total = this.total + x;
    return  this
  },
  this.sub = function (x) {
    this.total = this.total - x;
    return  this
  },
  this.increment = function () {
    this.total = this.total + 1;
    return this
  },
  this.decrement = function () {
    this.total = this.total - 1;
    return this
  },
  this.result = function () {
    console.log(this.total);
  }
}

var myCalc = new calculator();

myCalc.add(10).increment().increment().result();
```

5. imports

Imports are new functionality introduced in the ECMAScript 6.

It lets you import a function that has been created and exported in another JavaScript file.

Import the entire content of a module:

```
import mybasket from "module.js";
```

Import only a desired part of a module, and add a (shorter) alias:

```
import {longerModuleNameUnconvinientToUse as shortName} from "module.js";
```

You can use `export` in the origin file to tell the module which parts to use elsewhere:

file.js

```
function getJSON(url, callback) {
  let xhr = new XMLHttpRequest();
  xhr.onload = function () { callback(this.responseText) };
  xhr.open("GET", url, true);
  xhr.send();
}

export function getUsefulContents(url, callback) {
  getJSON(url, data => callback(JSON.parse(data)));
}
```

main.js

```
import {getUsefulContents as getContents} from "file.js";
getUsefulContents("http://www.example.com", data => {
  doSomethingUseful(data);
});
```

6. js api

We saw how to work with `geolocation`, `local storage` etc.

There is a huge, growing list of web api available for you to use.

Look through them and start using them!

MDN web APIs

7. 3D

http://threejs.org/