CITY UNIVERSITY
LONDON

Functions, Hosting this, bind, call, apply, => functions, let keyword!

Aris Markogiannakis
City University - Short Courses
CS3606 : JavaScript 2: Advanced Javascript for Websites and Web

This week?

This week we will learn about:

- Scope
- let/const keyword
- Arrow functions
- Hoisting
- This!
- Functions
- Bind/Call/Apply

Scopes

- There are two different types of Scope
  - **Global**
    - Variables created outside a function definition
  - **Functional**
    - Variables created with the function definition
  - **Block Scope**
    - Blocks can be if, for loops

```
var price = 200;

function setPrice() {
    var vat = 20;
}

console.log(price);
console.log(vat);
```

```
var students = 10;

function tutorial() {
    var tutor = "Aris";
}

tutorial();
console.log(students);
console.log(tutor);
```

Differences between functions and blocks…

- What will the result be for the second code example?

```
var students = 10;

function tutorial() {
    var tutor = "Aris";
}

tutorial();
console.log(students);
console.log(tutor);
```

```
var students = 10;

if (true) {
    var tutor = "Aris";
}

tutorial();
console.log(students);
console.log(tutor);
```

And this one

- In this example we have an IIFE, see below what is an IIFE.
- If you run the code you will notice that it is still behaves as a function

- An **IIFE**, or Immediately Invoked Function Expression, **is** a common JavaScript design pattern **used** by most popular libraries to place all library code inside of a local scope. In the words of James Padolsey: An **IIFE** protects a module's scope from the environment in which it **is** placed. (gregfranko.com/blog/i-love-my-iife)

```
var price = 20;

if (true){
    (function() {
        var vat = 20;

        console.log(price);
    }());
    console.log(price);
}
console.log(price);
```

```
var students = 20;

if (true) {
    (function() {
        students = 1
    })
}

console.log(students);
```

Finally this one

```
function calculateVAT() {
    var vat = 20;

    function doSomething() {
        var price = 20;
    }

    function doSomethingToo() {
        var fee = 20;

        if (typeof price !== "undefined") {
            fee = 12;
        }
        console.log(fee + vat);
    }
    doSomething();
    doSomethingToo();
}
```

The problem

Both “price” declarations point to the same variable

```
var price=20, fee=30;
if (fee>0) {
    var price = 50;
}
console.log(price);
```

The solution

The “c” variable declared with let is scoped to the if block

```
var c = 1, d = 2;
if (d > 0) {
  let c = 10;
  console.og(c); // 10
}
console.log(c); // 1
```

- Global scope – window object

```
// Global scope... not in a function
var e = 10;
console.log(window.e); // 10
```

- And with let?

```
// Global scope... not in a function
let k = 10;

console.log(window.k); // undefined
```

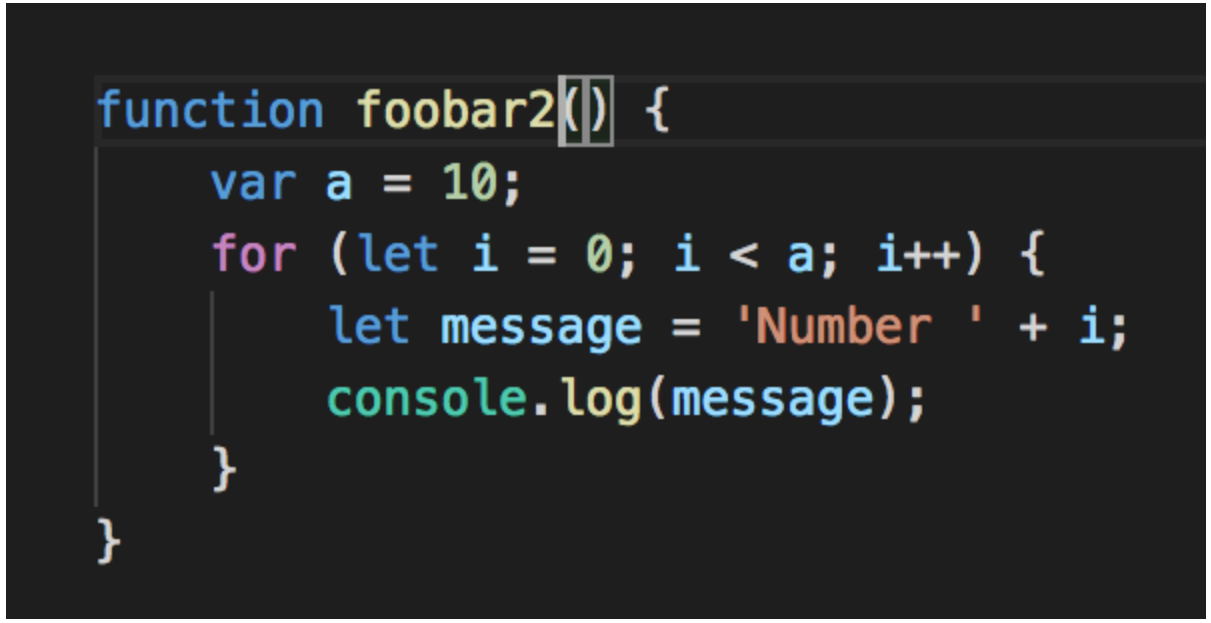
Bad

- The “i” and “message” variables are only used in the loop, but are available to entire function

```
function foobar() {
    var a = 10,
    message = '',
    i;
    for (i = 0; i < a; i++) {
        message = 'Number ' + i;
        console.log(message);
    }
}
```

Good

With let, we can declare “i” and “message” within the for block:
The “i” and “message” variables can only be seen inside the for loop



```
function foobar2() {
    var a = 10;
    for (let i = 0; i < a; i++) {
        let message = 'Number ' + i;
        console.log(message);
    }
}
```

Arrow functions

Why?

- make code less verbose
- resolve common issues with the this keyword

But?

- they can make it less readable
- They can resolve some of the problems we face with this
- But sometimes it is nice to have a dynamic this

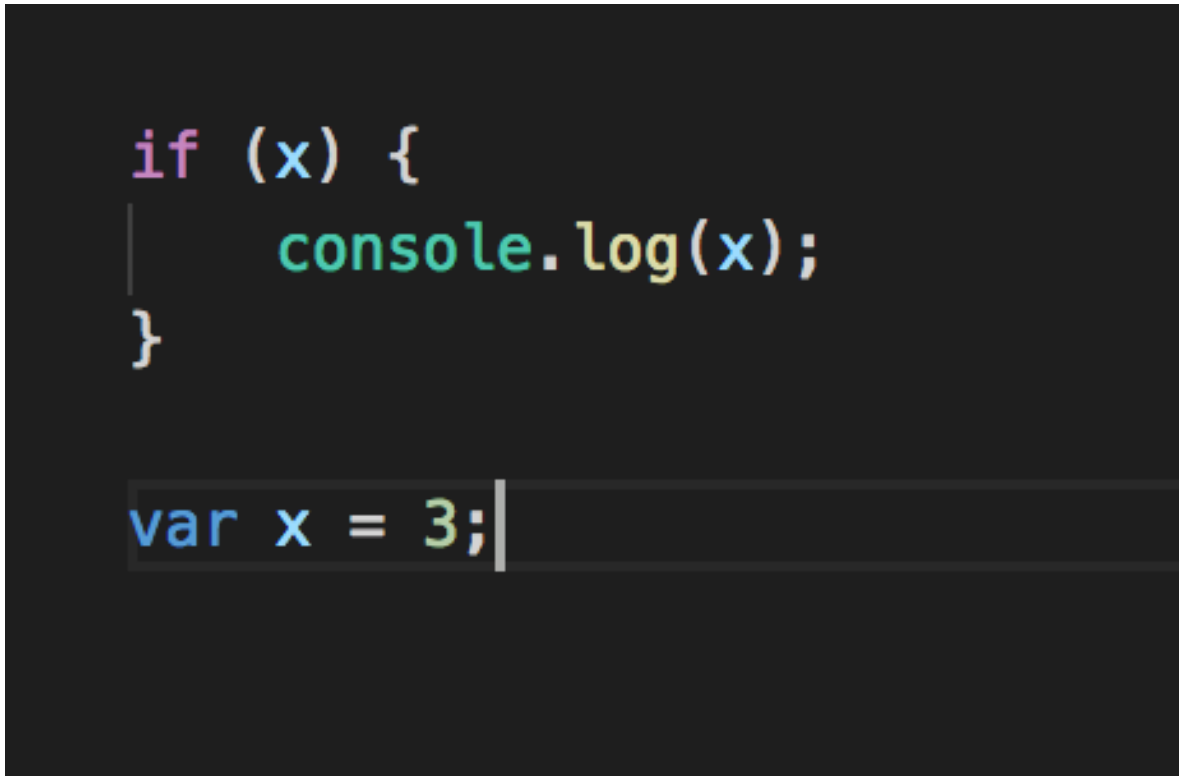
```
// A regular function definition
var timesTwo = function (value) {
    var result = value * 2;
    return result;
}
// its arrow function equivalent
var timesTwo = (value) => {
    var result = value * 2;
    return result;
}
```

Exercise 1

- Lets take a look on our exercises

Hoisting

- When evaluating your script, the JavaScript parser effectively moves any variable or function declarations to the top of their scope.



More on hoisting

```
function price() {
    vat();
    var price = 20;
}

// Will be parsed as
function price() {
    var price;
    vat();
    price = 20;
}
```

```
function price() {
    doSomething();
    var price=50;
    function myFunction() {/*Statements*/}
}

// Will be parsed as
function price() {
    var price;
    function myFunction() {/*Statements*/}
    doSomething();
    price=50;
}
```

Why hoisting

- Coding with hoisting in mind will avoid a lot of issues that can arise with variable and function declarations.

- It also helps us to understand how the order in which we write our scripts affects the end result

this - keyword

- When you create a function, the inner scope automatically receives a this keyword
- The value of this will vary, depending upon how the function is called.
  - and how the function was defined

Functions

- Functions are also objects
- Functions can be treated as a objects
- We can store functions in variable
- We can pass a function as an argument in a function **and** return a function from a function.

Functions returning functions (High Order Functions)

- On this example we can return a function from a function

```
// First class Functions: function returning functions

function sayHelloInLanguage(language){
    if (language === 'Greek'){
        return function(name) {
            console.log('Geia sou' +name);
        }
    } else if (language === 'Spanish') {
        return function(name) {
            console.log('Hola' +name);
        }
    } else {
        return function(name) {
            console.log('Hello' +name);
        }
    }
}

var englishHello = sayHelloInLanguage();
var spanishHello = sayHelloInLanguage('Spanish');
var greekHello = sayHelloInLanguage('Greek');

englishHello('Mike');
spanishHello('Mike');
greekHello('Mike');


sayHelloInLanguage('Spanish')('Bob');
```

Passing functions as arguments (High Order Functions)

- On this example we can pass a function as an argument.

```
// First class functions - passing functions as arguments

let prices = [ 1000, 2000, 3000];

console.log("hello");

function ManipulatePrices(arr, fn) {
    var result = [];

    for (var i=0; i<arr.length; i++) {
        result.push(fn(arr[i]));
    }

    return result;
}

let netValue = function(value, fee){
    return value - value*0.005;
}

let studentLoan = function(value, fee){
    return value - value*0.005;
}


let newValues = ManipulatePrices(prices, netValue );

console.log(newValues);


let studentLoans = ManipulatePrices(prices, netValue );

console.log(newValues);
```

IFEE

The common **advantage** of **IIFE** is that any "Function or Variable" defined inside **IIFE**, cannot be accessed outside the **IIFE** block, thus preventing global scope from getting polluted. Also helps us manage memory in an efficient manner.

We can only call it once – to initialise something and data privacy

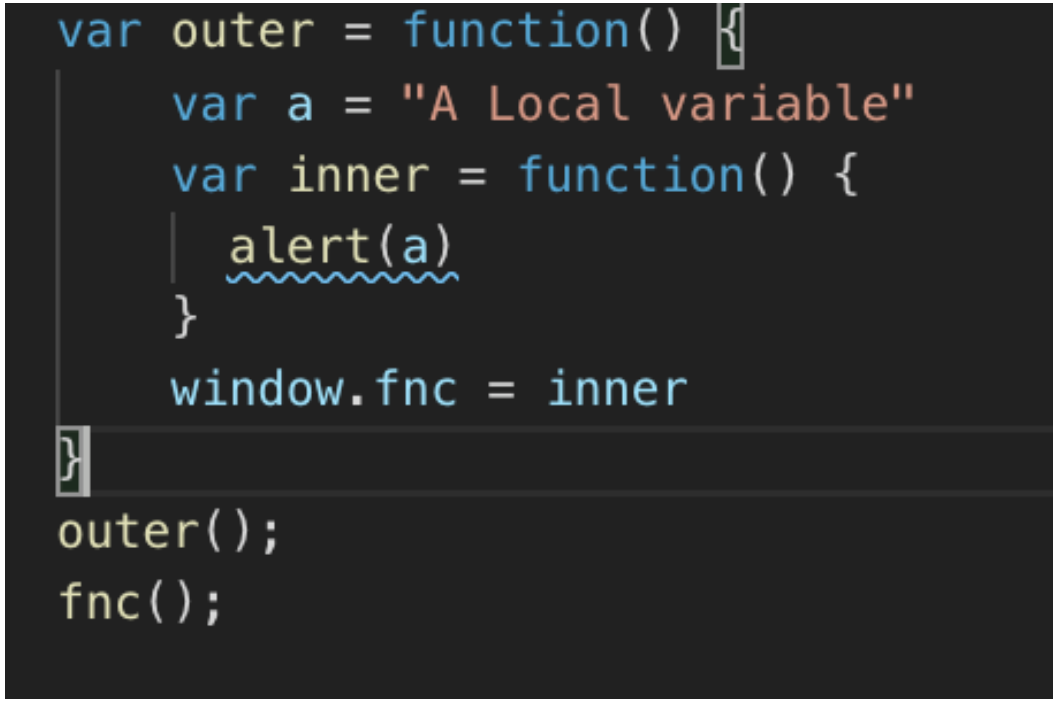
```
function CalculateDate() {
    var currentDate = new Date("23/10/2020");
    console.log(currentDate > new Date());
}

CalculateDate();


(function CalculateDate() {
    var currentDate = new Date("23/10/2020");
    console.log(currentDate > new Date());
}
)();
```

Closures

- The inner function has access to the local variables of the outer function.
- Normally when a function exits, all it's local variables are destroyed, but…
- If I hang on to the inner function in some way, the local variables of the outer function are still needed by the inner function.
- They are still in scope, so they are not garbage collected.



Exercise 2

- Conver the slide on 12 to closures

this: in functions

- Inside a function definition, this will also default to the global object

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

var result = getThis();
console.log(this);
```

- Inside an object method, the this keyword takes the value of the *object* the method belongs to.

```
var Product = {
    price: 20,
    buyMe: function () {
        return this;
    }
}

var buy = Product.buyMe();
```

call, apply, bind

- Call

With the call method, you pass an object as the first argument, followed by the target function arguments.

- Apply

With the apply method, you pass an object as the first argument, followed by the target function arguments as an array.

- Bind

With the bind method, you can create a new function where this will be a specified object

Call

```
function getFullPrice(fee, vat) {
    return this.price + this.countryRate + fee + vat;
}

var price = {
    price: 100,
    countryRate: 20
}

getFullPrice.call(price, 20, 50);
```

Apply (corrected)

```
function getFullPrice(fee, vat) {
    return this.price + this.countryRate + fee + vat;
}

var price = {
    price: 100,
    countryRate: 20
}

var fees = [20, 5]

getFullPrice.apply(price, fees);
```

Bind

```
var price = 20;
var product = {
    price: 20,
    getPrice:  function() {
        console.log(this.price);
    }
}

product.getPrice();

var newGetPrice = product.getPrice();
newGetPrice();

var BindGetPrice = product.getPrice.bind(product);
BindGetPrice();
```

What was the difference between call and apply?

- **Difference between call** and **apply** is just that **apply** accepts parameters **in the** form of an array while **call** simply can accept a comma separated list of arguments.
- A **bind** function is basically which binds the context of something and then stores it into a variable for execution at a later stage.

The let keyword

- As of 2015 a new way of declaring variables
- Not a replacement of var although in many ways it can be seen as the same

```
// Create a global variable called "a"
 var a = 1;

// or ..
let a = 1;
```

Differences between var and let

- var declarations are scoped to the function they appear in,
- let declarations are scoped to the code block they appear in
  - Code blocks – if, for , while, switch etc

- let declarations are hoisted in a slightly different manner to var declarations
- Var declarations in the global scope create a corresponding property on the window object, let declarations d

Finally…

- Time for exercise 3