ADVANCED JAVASCRIPT FOR WEB SITES AND WEB APPLICATIONS

Session 3

1. Functions and Scope
2. Hoisting
3. `this`

Functions and Scope

Here are three of main ways to create new functions:

Function declaration:

```
function foo () {
  // function body
}
```

Function expression:

```
var bar = function () {
  // function body
};
```

Named function expression:

```
var bar = function foo () {
  // function body
};
```

JavaScript does not have block scope.

In the language, only functions introduce a new scope.

Outside of any function, variable created belong to the global scope. They will be accessible throughout the script.

Variables created within a function, will only be accessible within that function (if they are created using the `var` keyword...).

```
var a = "hello";

function newVar () {
  var b = 1;
}

console.log(a); // hello
console.log(b); // throws a ReferenceError
```

Consider the following:

```
var a = "hello";

function gilles () {
  var a = 1;
}

console.log(a);
```

What does the console.log return?

the `console.log` returns "hello"

And in this case:

```
var a = "hello";

if (true) {
  var a = 1;
}

console.log(a);
```

What does the console.log return?

the `console.log` returns 1

Since only functions create a new scope, in this case, we are simply changing the value of the already defined variable `a`. And its value is changed thereafter.

But...

```
var a = "hello";

if(true) {

  (function () {
    var a = 1;
    console.log(a);
  }());

  console.log(a);
}

console.log(a);
```

this returns "1", "hello", "hello"

Using an immediately-invoked functions, we can create a temporary scope where any variable declared will not effect the outside scope.

Exercise 1

What does console log return?

```
var a = "hello";

if(true) {

  (function () {
    a = 1;
  }());

}
console.log(a);
```

console.log(a) returns "1"

Exercise 2

```
function func1 () {

  var a = 10;

  function func2 () {
    var b = 4;
    console.log(b + a);
  }
  func2(); // what does this display?

  function func3 () {
    var c = 7;
    if (typeof b !== "undefined") {
      c = 13;
    }
    else {
      c = 17;
    }
    console.log(c + a);
  }
  func3(); // what does this display?
}

func1();
```

Remember that `typeof` of a variable will return the type of variable if that variable exists in the current scope ("string", "boolean", "number" etc.). If the variable does not exist in the scope, it will return "undefined".

func2() prints 14

func3() prints 27

2. Hoisting

When running your script, the JavaScript parser moves any variable or function declaration to the top of the current scope.

For variables, only the declaration is hoisted to the top of the scope, not the value assignment.

This code:

```
function foo () {
  bar(); // call a function here
  // do something here
  // do something else here
  var a = 0;
}
```

Will be parsed as:

```
function foo () {
  var a;
  bar(); // call a function here
  // do something here
  // do something else here
  a = 0;
}
```

For function declarations, the entire declaration is hoisted to the top of the scope.

This code:

```
function foo () {
  bar(); // do something here
  var a = 0;
  function MyFunction () {
    // statement
  }
}
```

Will be parsed as:

```
function foo () {
  var a;
  function MyFunction () {
    // statement
  }
  bar(); // do something here
  a = 0;
}
```

For name function expressions, the name of the variable is hosited to the top, but not the function body or function name.

This code:

```
var gilles = "gilles";

var myFunction = function myFunctionPrivateName () {
  // statement
}
```

Is parsed as:

```
var gilles, myFunction;

gilles = "gilles";

myFunction = function myFunctionPrivateName () {
  // statement
}
```

It does not matter if a variable or function declaration is never used, it will still get hoisted to the top of the scope.

Example:

```
function foo () {
  bar(); // do something here
  if(false) {
    var c = "abc";
    function func () {
      //do something
    }
  }
}
```

```
function foo () {
  function func () {
    //do something
  }
  var c;
  bar(); // do something here
  if(false) {
    c = "abc";
  }
}
```

Coding with this in mind will avoid lots of issues around variable and function delcaration.

This is why it is recommended to move all variable declaration at the top of their scope. And to optimise this by using the single var pattern.

Exercise 3

what does the console log return? Or another way to look at this is: is test a global variable?

```
(function () {

  test = 5;

  if (false) {
    var test;
  } else {
    // do something here
  }

}());

console.log(test);
```

The console will return an error, test is a local variable to the current scope.

Exercise 4

what does the console log return?

```
(function () {
    test = 5;

    if (false) {
        var test = 3;
    } else {
        console.log(addOne(test));
    }

    function addOne(value) {
        return value + 1;
    }
}());
```

6

Exercise 5

what does the console log return?

```
(function () {
    test = 5;

    if (true) {
        var test = 3;
        console.log(addOne(test));
    }

    function addOne(value) {
        return value + 1;
    }
}());
```

4

3. `this`

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

The value of the 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.

```
console.log(this);
```

inside a function, as 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(); //returns the object "o"
```

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

call, apply, bind

When using a function's `call`, `apply` or `bind` method, you can chose the value of `this`.

Using `call`, you pass an object as the first argument to the method, along with the rest of the target function arguments.

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

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

add.call(o, 5, 7);
```

Using `apply`, you pass an object as the first argument to the method, along with the rest of the target function arguments as an array.

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

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

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

Using `bind` you can extract a function that has been set as an object method, and use it in another context.

```
this.x = 9;

var module = {
  x: 81,
  getX: function() { return this.x; }
};

module.getX(); // 81

var getX = module.getX;
getX(); // 9, because in this case, "this" refers to the global object

// Create a new function with 'this' bound to module
var boundGetX = getX.bind(module);
boundGetX(); // 81
```

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
```

Exercise 6

what do both console logs return?

```
var fullname = "John Doe";
var obj = {
   fullname: "Colin Ihrig",
   prop: {
      fullname: "Aurelio De Rosa",
      getFullname: function() {
         return this.fullname;
      }
   }
};

console.log(obj.prop.getFullname());

var test = obj.prop.getFullname;

console.log(test());
```

```
console.log(obj.prop.getFullname()); // returns "Aurelio De Rosa"

console.log(test()); // returns "John Doe"
```

Amend the previous `text` variable so it now returns "Aurelio De Rosa". Create a new variable and extract the value "Colin Ihrig" to it.

```
var test = obj.prop.getFullname.bind(obj.prop);

var test2 = obj.prop.getFullname.bind(obj);
```

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.

Build an object that has a "total" property (a number). The object also needs 5 methods:

- a method that adds a number to `total`. The number to add will be passed as an argument
- a method that substracts a number to `total`. The number to add will be passed as an argument
- a method that increments `total` by 1
- a method that decrements `total` by 1
- a method that console logs the value of `total`

You should then be able to do:

```
myObjectName.methodIncrement().methodAdd(10).methodSubstract(5);
myObjectName.result();
```

```
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);
    return this
  }
}

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