Chapter 56: Scope

Section 56.1: Closures

When a function is declared, variables in the context of its *declaration* are captured in its scope. For example, in the code below, the variable `x` is bound to a value in the outer scope, and then the reference to `x` is captured in the context of `bar`:

```
var x = 4; // declaration in outer scope

function bar() {
    console.log(x); // outer scope is captured on declaration
}

bar(); // prints 4 to console
```

> Sample output: `4`

This concept of "capturing" scope is interesting because we can use and modify variables from an outer scope even after the outer scope exits. For example, consider the following:

```
function foo() {
    var x = 4; // declaration in outer scope

    function bar() {
        console.log(x); // outer scope is captured on declaration
    }

    return bar;

    // x goes out of scope after foo returns
}

var barWithX = foo();
barWithX(); // we can still access x
```

> Sample output: `4`

In the above example, when `foo` is called, its context is captured in the function `bar`. So even after it returns, `bar` can still access and modify the variable `x`. The function `foo`, whose context is captured in another function, is said to be a *closure*.

Private data

This lets us do some interesting things, such as defining "private" variables that are visible only to a specific function or set of functions. A contrived (but popular) example:

```
function makeCounter() {
    var counter = 0;

    return {
        value: function () {
            return counter;
```

```
        },
        increment: function () {
            counter++;
        }
    };
}

var a = makeCounter();
var b = makeCounter();

a.increment();

console.log(a.value());
console.log(b.value());
```

> Sample output:
>
> 1 0

When `makeCounter()` is called, a snapshot of the context of that function is saved. All code inside `makeCounter()` will use that snapshot in their execution. Two calls of `makeCounter()` will thus create two different snapshots, with their own copy of `counter`.

Immediately-invoked function expressions (IIFE)

Closures are also used to prevent global namespace pollution, often through the use of immediately-invoked function expressions.

Immediately-invoked function expressions (or, perhaps more intuitively, *self-executing anonymous functions*) are essentially closures that are called right after declaration. The general idea with IIFE's is to invoke the side-effect of creating a separate context that is accessible only to the code within the IIFE.

Suppose we want to be able to reference `jQuery` with `$`. Consider the naive method, without using an IIFE:

```
var $ = jQuery;
// we've just polluted the global namespace by assigning window.$ to jQuery
```

In the following example, an IIFE is used to ensure that the `$` is bound to `jQuery` only in the context created by the closure:

```
(function ($) {
    // $ is assigned to jQuery here
})(jQuery);
// but window.$ binding doesn't exist, so no pollution
```

See the canonical answer on Stackoverflow for more information on closures.

Section 56.2: Hoisting

What is hoisting?

Hoisting is a mechanism which moves all variable and function declarations to the top of their scope. However, variable assignments still happen where they originally were.

For example, consider the following code:

```
console.log(foo);  // → undefined
var foo = 42;
console.log(foo);  // → 42
```

The above code is the same as:

```
var foo;           // → Hoisted variable declaration
console.log(foo);  // → undefined
foo = 42;          // → variable assignment remains in the same place
console.log(foo);  // → 42
```

Note that due to hoisting the above **`undefined`** is not the same as the `not defined` resulting from running:

```
console.log(foo);  // → foo is not defined
```

A similar principle applies to functions. When functions are assigned to a variable (i.e. a function expression), the variable declaration is hoisted while the assignment remains in the same place. The following two code snippets are equivalent.

```
console.log(foo(2, 3));     // → foo is not a function

var foo = function(a, b) {
    return a * b;
}
```

```
var foo;
console.log(foo(2, 3));     // → foo is not a function
foo = function(a, b) {
    return a * b;
}
```

When declaring function statements, a different scenario occurs. Unlike function statements, function declarations are hoisted to the top of their scope. Consider the following code:

```
console.log(foo(2, 3));  // → 6
function foo(a, b) {
    return a * b;
}
```

The above code is the same as the next code snippet due to hoisting:

```
function foo(a, b) {
    return a * b;
}

console.log(foo(2, 3));  // → 6
```

Here are some examples of what is and what isn't hoisting:

```
// Valid code:
foo();

function foo() {}

// Invalid code:
bar();                   // → TypeError: bar is not a function
```

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


// Valid code:
foo();
function foo() {
    bar();
}
function bar() {}


// Invalid code:
foo();
function foo() {
    bar();                // → TypeError: bar is not a function
}
var bar = function () {};


// (E) valid:
function foo() {
    bar();
}
var bar = function(){};
foo();
```

Limitations of Hoisting

Initializing a variable can not be Hoisted or In simple JavaScript Hoists declarations not initialization.

For example: The below scripts will give different outputs.

```
var x = 2;
var y = 4;
alert(x + y);
```

This will give you an output of 6. But this...

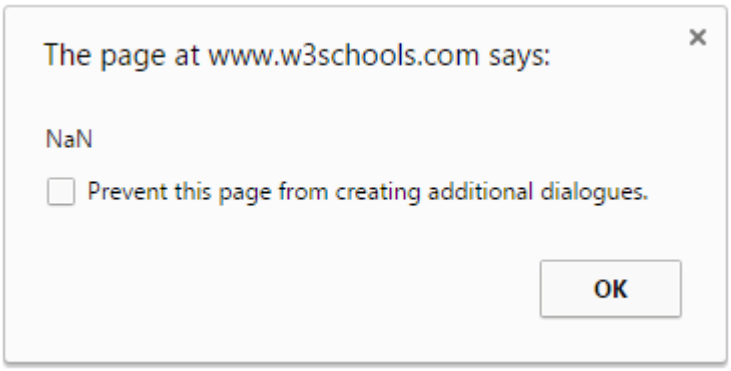
```
var x = 2;
alert(x + y);
var y = 4;
```

This will give you an output of NaN. Since we are initializing the value of y, the JavaScript Hoisting is not happening, so the y value will be undefined. The JavaScript will consider that y is not yet declared.

So the second example is same as of below.

```
var x = 2;
var y;
alert(x + y);
y = 4;
```

This will give you an output of NaN.



Section 56.3: Difference between var and let

(Note: All examples using `let` are also valid for `const`)

`var` is available in all versions of JavaScript, while `let` and `const` are part of ECMAScript 6 and only available in some newer browsers.

`var` is scoped to the containing function or the global space, depending when it is declared:

```
var x = 4; // global scope

function DoThings() {
    var x = 7; // function scope
    console.log(x);
}

console.log(x); // >> 4
DoThings();     // >> 7
console.log(x); // >> 4
```

That means it "escapes" `if` statements and all similar block constructs:

```
var x = 4;
if (true) {
    var x = 7;
}
console.log(x); // >> 7

for (var i = 0; i < 4; i++) {
    var j = 10;
}
console.log(i); // >> 4
console.log(j); // >> 10
```

By comparison, `let` is block scoped:

```
let x = 4;

if (true) {
    let x = 7;
    console.log(x); // >> 7
}

console.log(x); // >> 4

for (let i = 0; i < 4; i++) {
```

```
    let j = 10;
}
console.log(i); // >> "ReferenceError: i is not defined"
console.log(j); // >> "ReferenceError: j is not defined"
```

Note that `i` and `j` are only declared in the `for` loop and are therefore undeclared outside of it.

There are several other crucial differences:

Global variable declaration

In the top scope (outside any functions and blocks), `var` declarations put an element in the global object. `let` does not:

```
var x = 4;
let y = 7;

console.log(this.x); // >> 4
console.log(this.y); // >> undefined
```

Re-declaration

Declaring a variable twice using `var` doesn't produce an error (even though it's equivalent to declaring it once):

```
var x = 4;
var x = 7;
```

With `let`, this produces an error:

```
let x = 4;
let x = 7;
```

> TypeError: Identifier `x` has already been declared

The same is true when `y` is declared with `var`:

```
var y = 4;
let y = 7;
```

> TypeError: Identifier `y` has already been declared

However variables declared with let can be reused (not re-declared) in a nested block

```
let i = 5;
{
   let i = 6;
   console.log(i); // >> 6
}
console.log(i); // >> 5
```

Within the block the outer `i` can be accessed, but if the within block has a `let` declaration for `i`, the outer `i` can not be accessed and will throw a `ReferenceError` if used before the second is declared.

```
let i = 5;
```

```
{
    i = 6;  // outer i is unavailable within the Temporal Dead Zone
    let i;
}
```

> ReferenceError: i is not defined

Hoisting

Variables declared both with `var` and `let` are hoisted. The difference is that a variable declared with `var` can be referenced before its own assignment, since it gets automatically assigned (with `undefined` as its value), but `let` cannot–it specifically requires the variable to be declared before being invoked:

```
console.log(x); // >> undefined
console.log(y); // >> "ReferenceError: `y` is not defined"
//OR >> "ReferenceError: can't access lexical declaration `y` before initialization"
var x = 4;
let y = 7;
```

The area between the start of a block and a `let` or `const` declaration is known as the Temporal Dead Zone, and any references to the variable in this area will cause a `ReferenceError`. This happens even if the variable is assigned before being declared:

```
y=7; // >> "ReferenceError: `y` is not defined"
let y;
```

In non-strict-mode, assigning a value to a variable without any declaration, automatically declares the variable in the global scope. In this case, instead of `y` being automatically declared in the global scope, `let` reserves the variable's name (`y`) and does not allow any access or assignment to it before the line where it is declared/initialized.

Section 56.4: Apply and Call syntax and invocation

The `apply` and `call` methods in every function allow it to provide a custom value for `this`.

```
function print() {
    console.log(this.toPrint);
}

print.apply({ toPrint: "Foo" }); // >> "Foo"
print.call({ toPrint: "Foo" }); // >> "Foo"
```

You might notice that the syntax for both the invocations used above are the same. i.e. The signature looks similar.

But there is a small difference in their usage, since we are dealing with functions and changing their scopes, we still need to maintain the original arguments passed to the function. Both `apply` and `call` support passing arguments to the target function as follows:

```
function speak() {
    var sentences = Array.prototype.slice.call(arguments);
    console.log(this.name+": "+sentences);
}
var person = { name: "Sunny" };
speak.apply(person, ["I", "Code", "Startups"]); // >> "Sunny: I Code Startups"
speak.call(person, "I", "<3", "Javascript"); // >> "Sunny: I <3 Javascript"
```

Notice that `apply` allows you to pass an `Array` or the `arguments` object (array-like) as the list of arguments, whereas, `call` needs you to pass each argument separately.

These two methods give you the freedom to get as fancy as you want, like implementing a poor version of the ECMAScript's native `bind` to create a function that will always be called as a method of an object from an original function.

```
function bind (func, obj) {
    return function () {
        return func.apply(obj, Array.prototype.slice.call(arguments, 1));
    }
}

var obj = { name: "Foo" };

function print() {
    console.log(this.name);
}

printObj = bind(print, obj);

printObj();
```

This will log

> "Foo"

The `bind` function has a lot going on

1. `obj` will be used as the value of **`this`**
2. forward the arguments to the function
3. and then return the value

Section 56.5: Arrow function invocation

`Version ≥ 6`

When using arrow functions **`this`** takes the value from the enclosing execution context's **`this`** (that is, **`this`** in arrow functions has lexical scope rather than the usual dynamic scope). In global code (code that doesn't belong to any function) it would be the global object. And it keeps that way, even if you invoke the function declared with the arrow notation from any of the others methods here described.

```
var globalThis = this; //"window" in a browser, or "global" in Node.js

var foo = (() => this);

console.log(foo() === globalThis);          //true

var obj = { name: "Foo" };
console.log(foo.call(obj) === globalThis);  //true
```

See how **`this`** inherits the context rather than referring to the object the method was called on.

```
var globalThis = this;
```

```
var obj = {
    withoutArrow: function() {
        return this;
    },
    withArrow: () => this
};

console.log(obj.withoutArrow() === obj);      //true
console.log(obj.withArrow() === globalThis);  //true

var fn = obj.withoutArrow; //no longer calling withoutArrow as a method
var fn2 = obj.withArrow;
console.log(fn() === globalThis);             //true
console.log(fn2() === globalThis);            //true
```

Section 56.6: Bound invocation

The `bind` method of every function allows you to create new version of that function with the context strictly bound to a specific object. It is especially useful to force a function to be called as a method of an object.

```
var obj = { foo: 'bar' };

function foo() {
    return this.foo;
}

fooObj = foo.bind(obj);

fooObj();
```

This will log:

> bar

Section 56.7: Method invocation

Invoking a function as a method of an object the value of **`this`** will be that object.

```
var obj = {
    name: "Foo",
    print: function () {
        console.log(this.name)
    }
}
```

We can now invoke print as a method of obj. **`this`** will be obj

```
obj.print();
```

This will thus log:

> Foo

Section 56.8: Anonymous invocation

Invoking a function as an anonymous function, `this` will be the global object (`self` in the browser).

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

func() === window; // true
```

Version = 5

In ECMAScript 5's strict mode, `this` will be `undefined` if the function is invoked anonymously.

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

This will output

> `undefined`

Section 56.9: Constructor invocation

When a function is invoked as a constructor with the `new` keyword `this` takes the value of the object being constructed

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

var obj = new Obj("Foo");

console.log(obj);
```

This will log

> { name: "Foo" }

Section 56.10: Using let in loops instead of var (click handlers example)

Let's say we need to add a button for each piece of `loadedData` array (for instance, each button should be a slider showing the data; for the sake of simplicity, we'll just alert a message). One may try something like this:

```
for(var i = 0; i < loadedData.length; i++)
    jQuery("#container").append("<a class='button'>"+loadedData[i].label+"</a>")
        .children().last() // now let's attach a handler to the button which is a child
        .on("click",function() { alert(loadedData[i].content); });
```

But instead of alerting, each button will cause the

> TypeError: loadedData[i] is undefined

error. This is because the scope of `i` is the global scope (or a function scope) and after the loop, `i == 3`. What we need is not to "remember the state of `i`". This can be done using **`let`**:

```
for(let i = 0; i < loadedData.length; i++)
    jQuery("#container").append("<a class='button'>"+loadedData[i].label+"</a>")
        .children().last() // now let's attach a handler to the button which is a child
        .on("click",function() { alert(loadedData[i].content); });
```

An example of `loadedData` to be tested with this code:

```
    var loadedData = [
        { label:"apple",      content:"green and round" },
        { label:"blackberry", content:"small black or blue" },
        { label:"pineapple",  content:"weird stuff.. difficult to explain the shape" }
    ];
```

A fiddle to illustrate this