Advanced JavaScript

ITM - SDM S9

EPITA

Scope

What is the scope?

Scope is the set of rules that determines where and how a variable (identifier) can be looked-up

Three important entities :

1. ***Engine :*** charge of **compilation** and **execution** of JavaScript program.
2. ***Compiler*** : handles all the dirty work of parsing and code-generation.
3. ***Scope*** : ***collects and maintains a look-up list*** of all the declared variables, and enforces a strict set of rules as to how these are accessible to currently executing code.

What happens when the compiler sees `var a = 2;` statement ?

In fact, there are two statements:

1. `var a` which **Compiler** handle during compilation
2. `a = 2` which **Engine** handle during exection

Compiler will instead proceed as:

1. **Compiler** asks to if variable a exists. If so, **Compiler** ignores this declaration and moves on. If not, **Compiler** asks to **Scope** to declare a new variable called `a` for that scope declaration.

2. **Compiler** produces code for **Engine**, to handle `a = 2`. The **Engine** begin to asks to **Scope** if there is a varibale called `a` in the current scope. If so, **Engine** uses variable. If not, Engine looks elsewhere

Lexical Scope

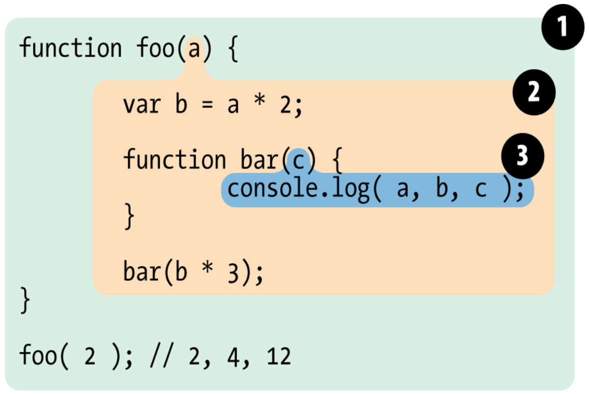
What is the lexing ?

The lexing process examines a string of source code characters and assigns semantic meaning to the tokens as a result of some stateful parsing.

What is the lexical scope ?

Lexical scope is based on where variables and blocks of scope are authored, by you, at write time.

Nested Scope



```
function foo(a) {

        var b = a * 2;

        function bar(c) {
                console.log( a, b, c );
        }

        bar(b * 3);
}

foo( 2 ); // 2, 4, 12
```

Bubble 1 covered the global scope which includes identifier: `foo`.

Bubble 2 covered the scope of `foo` which includes identifiers: `bar`, `a` and `b`.

Bubble 3 covered the scope of `bar` which includes identifier: `c`.

In this above code snippet, the ***Engine*** executes the `console.log(...)` and goes looking for the three referenced variables `a`, `b`, and `c`. It begins with innermost scope bubble, the scope of the `bar(..)` function. It won't find `a` there, so it goes up one level, out to the next nearest scope bubble, the scope of `foo(..)`. It finds `a` there, and so it uses that `a`. Same thing for `b`. `c` is found inside scope of `bar`.

Hoisting

```
a = 2;

var a;

console.log( a );
```

What do you expect ?

And in this piece of code ?

```
console.log( a );

var a = 2;
```

Why ?

Variables and function are processed first before any part of your is executed

```
a = 2;

var a;

console.log( a );
```

We see this.

```
/* Compilation time */

var a;


/* Execution time */

a = 2;

console.log( a );
```

JavaScript sees that.

```
console.log( a );

var a = 2;
```

We see this.

```
/* Compilation time */

var a;


/* Execution time */

console.log( a );

a = 2;
```

JavaScript sees that.

It's also important to note that hoisting is per-scope

```
function foo() {
  var a;

  console.log( a ); // undefined

  a = 2;
}

foo();
```

The program is interpreted like this :

```
function foo() {
  var a;

  console.log( a ); // undefined

  a = 2;
}

foo();
```

Function expressions are not hoisted only function declarations

```
foo(); // TypeError!

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

Function first

```
foo(); // 1

var foo;

function foo() {
  console.log( 1 );
}

foo = function() {
  console.log( 2 );
};
```

The snippet code is interpreted by JavaScript like this:

```
function foo() {
  console.log( 1 );
}

foo(); // 1

foo = function() {
  console.log( 2 );
};
```

Closure

Definition

Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.

```
function foo() {
    var a = 2;

    function bar() {
        console.log( a ); // 2
    }

    bar();
}

foo();
```

In the above snippet code function `bar()` has a ***closure*** over the scope of `foo()`

Demonstration of *closure*

```
function foo() {
  var a = 2;

  function bar() {
    console.log( a );
  }

  return bar;
}

var baz = foo();

baz(); // 2 -- Whoa
```

bar() still has a reference to that scope, and that reference is called closure.

```
for (var i=1; i<=3; i++) {
    setTimeout( function timer(){
        console.log("Print : " + i);
    }, i*1000 );
}
```

What is display on the JavaScript console ?

Print : 4

Print : 4

Print : 4

Why ?

The timeout function callbacks are running well after the completion loop.

What do we want ?

We want that each iteration of loop captures its own copy of variable i.

IIFE(Immediately Invoked Function Expression)

```
for (var i=1; i<=3; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log("Print : " + j);
        }, j*1000 );
    })();
}
```

With `let`:

```
for (let i=1; i<=3; i++) {
        setTimeout(function timer(){
            console.log("Print : " + j);
        }, j*1000 );
}
```

With `let` keyword variable will be declared not just once for the loop, **but each iteration.**

Modules

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Objects

Syntax

Javascript object can be created by two forms:

1. Declarative form:

```
var myObj = {
  key: value
}
```

1. Constructed form:

```
var myObj = new Object();

myObj.key = value;
```

Built-in Objects

There are several other object sub-types:

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`
- `RegExp`
- `Error`

This objects can be used as a constructor with a `new` operator:

```
var strOjbect = new String("I am a string")
```

Object coercion

```
var strPrimitive = "I am string"; // this is a string primitive.
```

To perform operation on primitive type a built-in objects are required, in our case "String" if we want to realize operation like checking its length.

Fortunately, the language automatically coerces a `"string"` primitive to a `String` object for perform operation like checking length of string. So, you almost never need to use create Object form.

```
var strPrimitive = "I am string"; // this is a string primitive.
console.log( strPrimitive.lenght );      // 11
```

Only use the constructed form if you need the extra options.

Content

```
var myObject = {
  a: 2
};

myObject.a;  //2

myObject[a]; //2
```

The engine JavaScript doesn't store the value of property but it stores the property name which act as pointers to where the values are stored.

To access the values of properties object, we can use either the `.` operator or the `[]` operator.

The main difference between two syntax is that `.` operator requires an `Identifier` compatible name after it, whereas the `[]` syntax can take basically any UTF-8/Unicocde compatible string as the name property.

```
var myObject = {
  "Weird-Key?": 2
};

myObject["Weird-Key?"];  // 2
```

With `[]`, we can programtically build up the value of string, such as:

```
  var wantA = true;
  var myObject = {
  a: 2
  };

  var idx;

  if (wantA) {
  idx = "a";
  }

  // later

  console.log( myObject[idx] ); // 2
```

With also can compute property names, like this:

```
var prefix = "foo";

var myObject = {
[prefix + "bar"]: "hello",
[prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

Arrays

In Javascript, arrays are objects that have some array-like characteristics.

Array Literals

Array Literal is an pair of square brackets surrounding zero or more values separated by commas.

The first value will get the property name '0', the seconde value will get the property name '1' and so on.

```
var myArray = [];

var myNumbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'height', 'nine'
];

console.log( myArray[1] ); // undefined
console.log( myNumbers[1] ); // one


console.log( myArray.length ); // 0
console.log( myNumbers.length ); // 10
```

But in Javascript we can get similar result with an object:

```
var myNumbersObject = {
'0': 'zero', '1': 'one', '2': 'two',
'3': 'three', '4': 'four', '5': 'five',
'6': 'six', '7': 'seven', '8': 'height',
'9': 'nine'
};

console.log(myNumbersObject['0']); // zero
```

But there are also significant diffrences. `Array` has a set of useful methods that `Object` does not have.

JavaScript allows an `array` to contain any mixtures of values:

```
var misc = [
  'string', 30, true, false, undefined,
  {propertyObject: '44'}, ['nested', 'array']
];

console.log( misc.length );
```

Length

Javascript's array `length` is not an upper bound. If you store an element with a subscript that is greater than or equal to the current `length` will increase to contain the new element.

```
var myArray = [];
myArray.length;   // 0

myArray[1000000] = true;
myArray.length  // 1000001
```

The `length` of array can be set explicitly. Making `length` larger does not allocate more space for the array. Making the `length` will cause all properties with a subscript that is greater than or equal to the new length to be deleted:

```
var myNumbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'height', 'nine'
];

myNumbers.length = 3;

// myNumbers is ['zero', 'one', 'two']
```

Add

We can add new element with method `push` :

```
var myNumbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'height', 'nine'
];

myNumbers.push('ten'); // ["zero", "one", "two", "three",
// "four", "five", "six", "seven", "height", "nine", "ten"]
```

Delete

We can delete array with method `splice`:

```
var myNumbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'height', 'nine'
];

myNumbers.splice(9, 1); // ["zero", "one", "two", "three",
// "four", "five", "six", "seven", "height"]
```

Enumeration

We can enumerate array with a `for`:

```
var myNumbers = [
'zero', 'one', 'two', 'three', 'four',
'five', 'six', 'seven', 'height', 'nine'
];

for(var i = 0; i < myNumbers.length; i++) {
  console.log(i);
}
```

Duplicating Objects:

With can deeply duplicate a object in Javascript with JSON or `Object.assign()`:

```
var myObject = {
    a: 'a',
    array: ['1', '2', '3', '4', '5'],
    object: {
        myFunction: function() {
            console.log('Hello');
        },
        string: 'Daddy'
    }
}

console.log(myObject);


var newMyObject = Object.assign({}, myObject)

var newMyObjectJSON = JSON.parse(JSON.stringify(newMyObject))

console.log(newMyObject);

console.log(newMyObjectJSON);
```