Școala informală de IT

ECMAScript 6 / 2015

Agenda

- Javascript Standards
- ECMAScript 2015 features
- Scoping, let and const
- Arrow functions
- Classes

Javascript Standards

Javascript Standards

- Javascript is standardized at ECMA International (the European association for standardizing information and communication systems) since 1996
- ECMAScript: a language standardized by ECMA International and overseen by the TC39 committee. The standard itself
- JavaScript: the name used for implementations of the ECMAScript standard

Javascript Standards

- Javascript supports all functionality outlined in ECMAScript Specification
- ECMAScript 6, also known as ECMAScript 2015
  - ~ Latest version of ECMAScript
  - The first update to the language since ES5 (2009)
  - Is partially (but more and more) implemented in browsers (compatibility table)

ECMAScript 6 Features

- **Complete list: http://www.ecma-international.org/ecma-262/6.0/**
- **Will discuss today:**
  - **Scoping, let and const**
  - **Arrow functions**
  - **Classes**

Scoping, let & const

Variable hoisting

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

undefined vs ReferenceError

```
console.log(typeof variable); // Output: undefined
```

In JavaScript, an undeclared variable is assigned the value undefined at execution and is also of type undefined.

```
console.log(variable); // Output: ReferenceError: variable is not defined
```

In JavaScript, a ReferenceError is thrown when trying to access a previously undeclared variable.

Hoisting variables

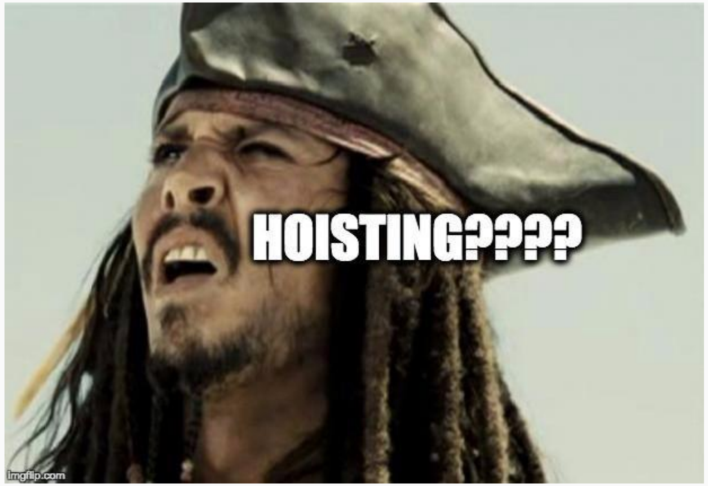
The scope of a variable declared with the keyword var is its current execution context. This is either the enclosing function or for variables declared outside any function, global.

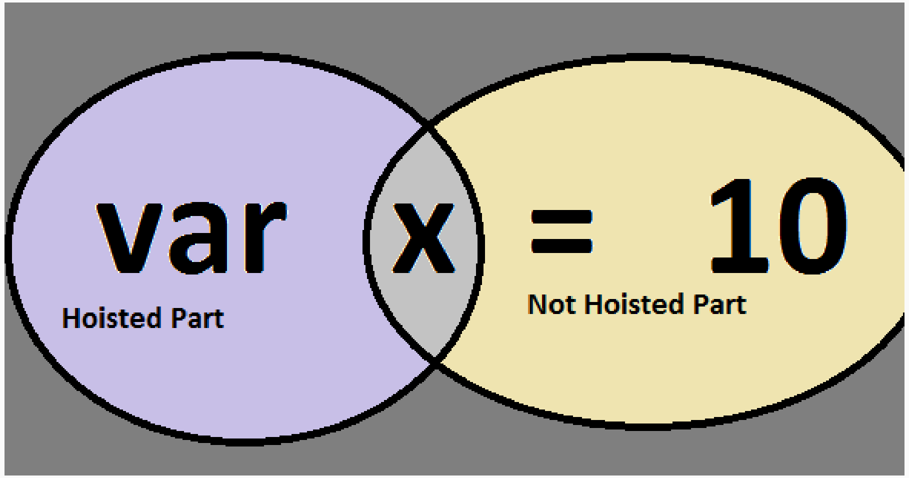
Let’s see what hoisting does

```
console.log(hoist); // Output: undefined

var hoist = 'The variable has been hoisted.';
```

And this is what JS does under the hood:

```
var hoist;

console.log(hoist); // Output: undefined

hoist = 'The variable has been hoisted.';
```





Inevitably, this means that no matter where functions and variables are declared, they are moved to the top of their scope regardless of whether their scope is global or local.

Of note however, is the fact that the hoisting mechanism only moves the declaration. The assignments are left in place.

If you've ever wondered why you were able to call functions before you wrote them in your code, that is the reason.

Let & const

- Let and const
  - Let declares a block-scoped local variable
  - Const declares a block-scoped constant
- Block scope
  - variables and constants are limited in scope to the block, statement or expression on which they are used
  - unlike var, which defines a variable globally or locally to an entire function regardless of block scope

Let vs var

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  //same var!
    console.log(x); // 2
  }
  console.log(x);  // 2
}
```

```
function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // different
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

Const

- **Rules:**
  - **The value of a constant cannot change through re-assignment**
  - **A constant can't be redeclared**
- **The rules apply only to the variable itself, not its assigned content**
  - **If the content is an object, the object itself can still be altered**

```
const PI = 3.1416;
PI = 3.141593; // err
const PI = 3.141594; // err
```

```
const OBJ = { ‘foo’: ‘bar’ };
OBJ = { ‘foo’: ‘bar2’ }; //err
OBJ.foo = ‘bar2’; // works
```

Arrow functions

Arrow functions

- Shorthands for functions using the => syntax
- Support
    - Expression bodies
    - Statement block bodies
- Arrow functions share the same lexical this as their surrounding code

Expression Bodies

```
const sum = function(x, y) {
  return x + y;
}
```

```
const sum = (x, y) => { return x + y; }
```

```
const sum = (x, y) => x + y;
```

Expression Bodies

```
const double = function(x) {
  return 2*x;
}
```

```
const double = (x) => { return  2*x; }
    or
const double = x => { return  2*x; }
```

```
const double = (x) => 2*x;
        or
const double = x => 2*x;
```

Statement Bodies

```
const numbers = [ 1, 2, 3, 4, 5 ];
const evens = [];
```

```
numbers.forEach(function(nr) {
        if (nr % 2 == 0) { evens.push(nr); }
});
```

Statements have to be put in braces:

```
numbers.forEach(nr => {
        if (nr % 2 == 0) { evens.push(nr); }
});
```

This in arrow functions: lexical this

```
function Timer() {
 this.value = 0;
 setInterval(function incr() {
   this.value++;
 }, 1000);
}
```

```
function Timer() {
 var that = this;
 that.value = 0;
 setInterval(function incr() {
   that.value++;
 }, 1000);
}
```

ES6:

```
function Timer() {
 this.value = 0;
 setInterval(() => {
   this.value++;
 }, 1000);
}
```

Returning object literals

```
const getNumberDetails = function(x) {
  return {
   “even”: x % 2 == 0 ? true : false
  }
}
```

```
const getNumberDetails = (x) => {
   “even”: x % 2 == 0 ? true : false
  }
```

✗

```
const getNumberDetails = (x) => ({
   “even”: x % 2 == 0 ? true : false
  })
```

✓

Syntax errors

```
const sum = (x, y)
=> { return x + y; }
```

✗

```
const sum = (x, y) =>
{ return x + y; }
```

✓

```
const sum = (x, y)
=> x + y;
```

✗

```
const sum = (x, y) =>
x + y;
```

✓

Classes

Classes

- Syntactic sugar over JavaScript’s prototype-based inheritance
- Do not introduce a new OOP inheritance model to JavaScript
- Class syntax:
    - Class declarations
    - Class expressions

Class declarations

```
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

Function declarations are hoisted, but class declarations are not.

```
const p = new Person(); // Reference Error

class Person {}
```

Class expressions

```
const Person = class {   // unnamed
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

Class expressions are not hoisted.

```
const Person = class Person { //named
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

Class body and method definitions

- Constructor
  - Special method for creating and initializing an object created with a class
  - Only one constructor per class
- Methods
  - Prototype methods
  - Static methods

Prototype methods - getter

```
const Person = class {
  constructor(fName, lName) {
    this.firstName = fName;
    this.lastName = lName;
  }
  get fullName() {
    return this.computeFullName();
  }
  computeFullName() {
    return this.firstName + ‘ ‘ + this.lastName;
  }
}
```

```
const p =
  new Person(‘Ken’, ‘Lee’);

console.log(p.fullName);
```

Prototype methods - setter

```
const Person = class {
  constructor() {
    firstName: 'Jimmy',
    lastName: 'Smith'
  }
  get fullName() {
      return this.firstName + ' ' + this.lastName;
  },
  set fullName (name) {
     var words = name.toString().split(' ');
     this.firstName = words[0] || '';
     this.lastName = words[1] || '';
  }
}
```

```
const p = new Person();
p.fullName = 'Jack Franklin';
console.log(p.firstName); // Jack
console.log(p.lastName) // Franklin
```

Static methods

- Are called without instantiating the class
- Cannot be called through a class instance
- Are often used to create utility functions for an application (see next slide)

```
const Person = class {
  …
  static defaultFullName() {
    return “John Doe”;
  }
}
```

Static methods

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  static distance(a, b) {
    var dx = a.x-b.x;
    var dy = a.y-b.y;
    return Math.sqrt(dx*dx + dy*dy);
  }
}
```

```
const p1 = new Point(2,3);
const p2 = new Point(7,4);

console.log(
  Point.distance(p1, p2)
);
```

Inheritance using extends

- extends is used to create a class as a child of another class
- if the subclass has a constructor, it needs to call super() before using "this"

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name +
' makes a noise.');
  }
}
```

```
class Dog extends Animal {
  speak() {
    console.log(this.name +
' barks.');
  }
}

const d = new Dog('Mitzie');
d.speak();
```

Resources

https://benmccormick.org/2015/09/14/es5-es6-es2016-es-next-whats-going-on-with-javascript-versioning/

https://github.com/lukehoban/es6features/blob/master/README.md

https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions

https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let