wantsome
The friendly IT Academy

Saptamana 8

Partea 1

Programare Front-End

1. JavaScript – ES6+, New Features

Syntactic sugar and new features

Versiunile **ES6+** ale JavaScript ofera anumite functionalitati de limbaj noi si modalitati mai usoare de folosire a unora deja existente - **syntactic sugar**

- Nu toate *browser*-ele sunt capabile sa inteleaga noile functionalitati (pentru ca nu sunt inca implementate in *kernel*-ul *browser*-ului)

- Codul JavaScript scris in varianta ES6 poate fi transformat in varianta ES5 (*foloseste doar functionalitati existente in ES5*) prin intermediul unui **transpiler**

- Un transpiler intelege codul scris intr-un anumit limbaj si-l poate transforma in alt limbaj dorit (in cazul nostru, se trece de la o versiune noua de JS la una veche)

  Exemplu: BABEL: ES6+ -> ES5 - https://babeljs.io/repl

ES6: declaring vars using **let** and **const** keywords

let si **const** - 2 keyword-uri noi folosite pentru declararea variabilelor; ambele se comporta diferit fata de **var**:

var - function scoped

```
var developer = "hey hi";
 function myFunction() {
     var hello = "hello";
 }
 console.log(hello);
 // error: hello is not defined
```

```
var greeter = "hey hi";
var times = 4;
if (times > 3) {
    var greeter = "say Hello instead";
}
console.log(greeter) //"say Hello instead"
```

ES6 let and const keywords

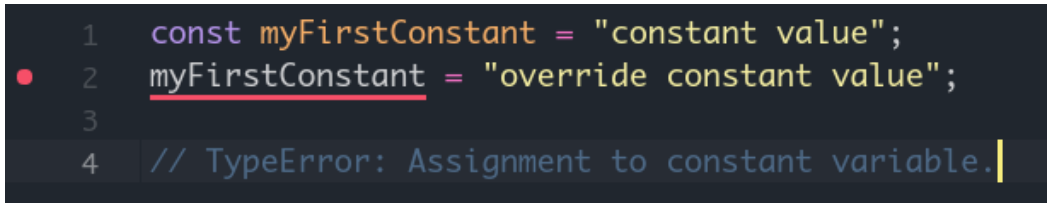
let - block scoped

```
let greeting = "say Hi";
if (true) {
    let greeting = "say Hello instead";
    console.log(greeting);//"say Hello instead"
}
console.log(greeting);//"say Hi"
```

ES6 let and const keywords

const

- block scoped
- folosit pentru a declara **valori constante** ! (*nu isi pot schimba valoarea in timp*)



reference let const

var let const

wantsome
The friendly IT Academy

Quiz
JavaScript – ES6

1. ***Quiz 1***

   let x = 2;
   let x = 'hello';
   console.log(x); /// ???
   a. hello
   b. 2
   c. undefined
   d. Uncaught SyntaxError: Identifier 'x' has already been declared

2. **Quiz 2**

   const x;
   x = 1;
   console.log(x); /// ???
   a. 1
   b. 'x'
   c. undefined
   d. Uncaught SyntaxError: Missing initializer in const declaration

Ex - http://bit.do/letEx

2. JavaScript - Scope & Scope Types

Scope

Scope se refera la un set de reguli care determina unde si cum poate fi o variabila referentiata

La baza JS, avem trei tipuri de *scope*:

- **Global Scope**
- **Local Scope / Function Scope**
- **Block Scope**

Variabilele create in interiorul unei functii sunt in ***local scope***, iar cele create in afara functiilor in ***global scope***.

Fiecare functie, atunci cand este invocata, creeaza un nou **scope.**

Scope Types

Global Scope

Variabilele declarate in scopul global pot fi accesate in oricare alt *scope*

```
var name = 'Wantsome';
console.log(name); // logs 'Wantsome'
function logName() {
    console.log(name); // 'name' is accessible here and everywhere else
}
logName(); // logs 'Wantsome'
```

Scope Types

Local Scope

Variabilele declarate in interiorul functiilor, pot fi accesate doar local (in interiorul lor)

```
// Global Scope
function someFunction() {
    // Local Scope #1
    function someOtherFunction() {
        // Local Scope #2
    }
}

// Global Scope
function anotherFunction() {
    // Local Scope #3
}
// Global Scope
```

Scope Types

Block Scope

- block statements (if - else, for, switch, while) - **NU** creaza un nou scope contrar asteptarilor
- pentru a declara variabile accesibile doar la nivel de *block,* putem folosit **let** si **const**

```
if (true) {
    // this 'if' conditional block doesn't create a new scope
    var name = 'Wantsome'; // name is still in the global scope
}
console.log(name); // logs 'Wantsome'
```

```
let greeting = "say Hi";
if (true) {
    let greeting = "say Hello instead";
    console.log(greeting);//"say Hello instead"
}
console.log(greeting);//"say Hi"
```

Scopes - Notions

Lexical Scope, Nested Scopes

- reprezinta un grup ‘nested’ de functii unde copiii au acces la variabilele declarate in parinte - NU SI INVERS !

```
function grandfather() {
    var name = 'Hammad';
    // likes is not accessible here
    function parent() {
        // name is accessible here
        // likes is not accessible here
        function child() {
            // Innermost level of the scope chain
            // name is also accessible here
            var likes = 'Coding';
        }
    }
}
```

3. Hoisting

Hoisting

Este un mecanism care sta la baza JavaScript si se refera la faptul ca anumite tipuri de declaratii ale functiilor si variabilelor sunt mutate in top-ul (la inceputul) scope-ului lor, inainte de momentul executiei.

Indiferent unde declaram functiile si variabilele, ele vor fi mutate la inceputul *scope*-ului lor, fie el global sau local.

Variables hoisting

```
console.log(hoist); // Output: undefined
var hoist = 'The variable has been hoisted.';
```

```
var hoist;

console.log(hoist); // Output: undefined
hoist = 'The variable has been hoisted.';
```

Hoisting

Function scoped variables

```
function hoist() {
  console.log(message);
  var message='Hoisting is all the rage!'
}

hoist();
```

```
function hoist() {
  var message;
  console.log(message);
  message='Hoisting is all the rage!'
}

hoist(); // Ouput: undefined
```

Hoisting

Strict mode

In ES5 putem evita erorile de hoisting cu ajutorul **strict mode**-ului declarat la inceputul fisierului javascript

```
'use strict';

console.log(hoist); // Output: ReferenceError: hoist is not defined
hoist = 'Hoisted';
```

Folosind **ES6** keywords precum **let** si **const** obtinem acelasi efect.

Hoisting

Hoisting Functions

1. Function declarations
2. Function expressions

Function declarations

```
hoisted(); // Output: "This function has been hoisted."

function hoisted() {
  console.log('This function has been hoisted.');
};
```

Function expressions

```
expression(); //Output: "TypeError: expression is not a function

var expression = function() {
  console.log('Will this work?');
};
```

[hoisting]

Hoisting

Order of precedence

Variable assignment > function declaration > variable declaration

Highly recommended - READ THIS BOOK

Hoisting example-exercise - http://bit.do/exHoist

Closures

- Un **closure** este intalnit in momentul in care o functie isi poate aminti scopul in care a fost creata si poate accesa valori ale variabilelor care au fost declarate in cadrul acestuia, cu toate ca functia in cauza se executa in afara acestui scop

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

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

  bar();
}

foo();
```

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

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

  return bar;
}

var baz = foo();

baz(); // returns 2 - WOAH, CLOSURE
```

[Closures explained]

Closure

Avem 3 aspecte pentru un *Closure*

1. Are acces la propriul scope (variabilele declarate in interiorul ei)
2. Are acces la variabilele dinafara functiei / a functiei ce o incapsuleaza (variabilele din outer function)
3. Are acces la variabilele globale

Un closure pastreaza scope-ul in momentul executiei functiei astfel incat are acces la variabilele create

Examples : https://codepen.io/oviduzz/pen/KYgGrL?editors=0011

https://codepen.io/oviduzz/pen/yrVBOQ?editors=1010

Fast quiz - http://bit.do/quizHoist

PRACTICE:
http://bit.do/exHoistClosure