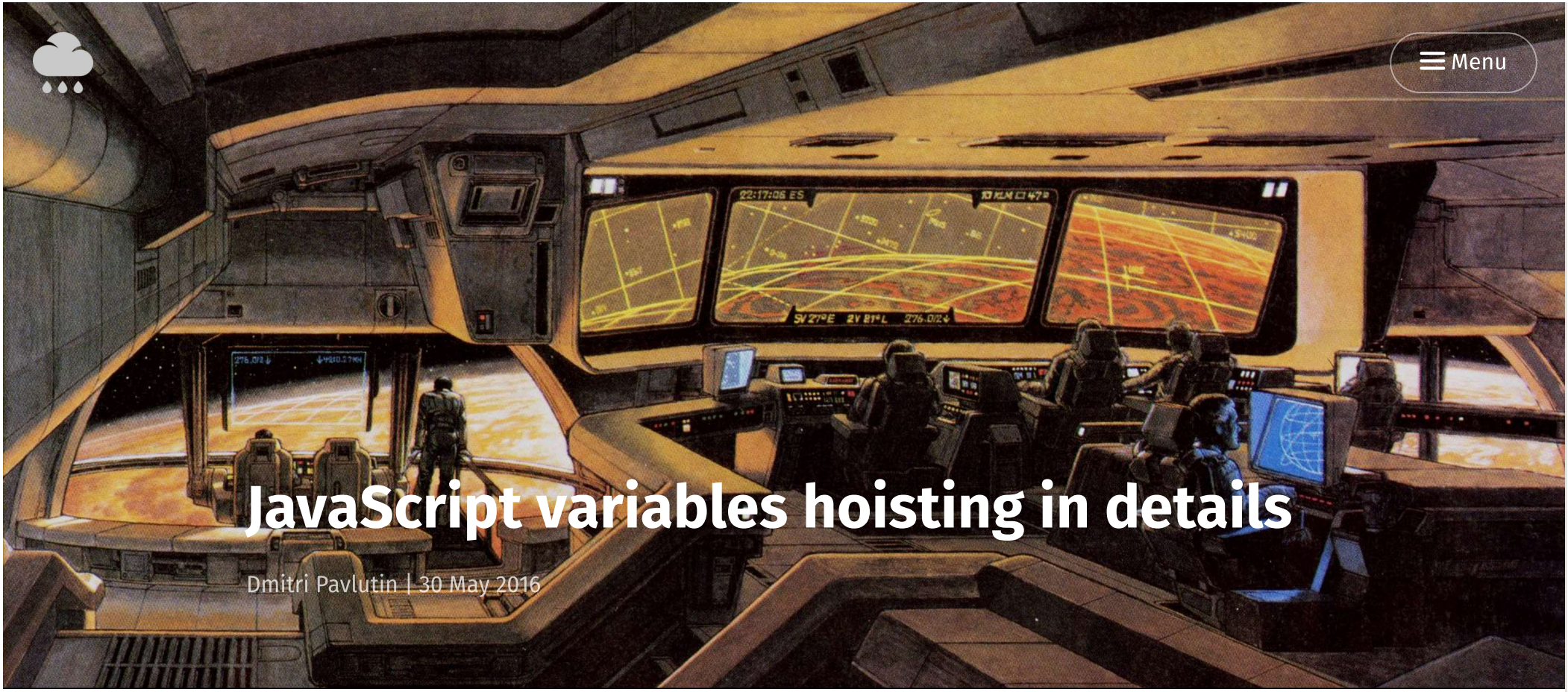

Menu

JavaScript variables hoisting in details

Dmitri Pavlutin | 30 May 2016

Variables in a program are everywhere. They are small pieces of data and logic that always interact with each other: and this activity makes the application alive.

In JavaScript an important aspect of working with variables is hoisting, which defines when a variable is accessible. If you're looking for a detailed description of this aspect, then you're in the right place. Let's begin.

1. Introduction

Hoisting is the mechanism of moving the variables and functions declaration to the top of the function scope (or global scope if outside any function).

Hoisting influences the variable life-cycle, which consists of these 3 steps:

* **Declaration** - create a new variable. E.g. `var myValue`
* **Initialization** - initialize the variable with a value. E.g. `myValue = 150`
* **Usage** - access and use the variable value. E.g. `alert(myValue)`

The process usually goes this way: first a variable should be *declared*, then *initialized* with a value and finally *used*. Let's see an example:

Try in JS Bin

```
// Declare
var strNumber;
// Initialize
strNumber = '16';
// Use
parseInt(strNumber); // => 16
```

A function can be *declared* and later *used* (or invoked) in the application. The *initialization* is omitted. For instance:

Try in JS Bin

```
// Declare
function sum(a, b) {
  return a + b;
}
// Use
sum(5, 6); // => 11
```

Everything looks simple and natural when these steps are successive: *declare* -> *initialize* -> *use*. If possible, you should apply this pattern when coding in JavaScript.

JavaScript does not follow strictly this sequence and offers more flexibility.
For instance, functions can be used before the declaration: *use* -> *declare*.
The following code sample first calls the function `double(5)`, and only later declares it `function double(num) {...}`:

Try in JS Bin

```
// Use
double(5); // => 10
// Declare
function double(num) {
  return num * 2;
}
```

It happens because the function declaration in JavaScript is hoisted to the top of the scope.

Hoisting affects differently:

* variable declarations: using `var`, `let` or `const` keywords
* function declarations: using `function <name>() {...}` syntax
* class declarations: using `class` keyword

Let's examine these differences in more details.

2. Function scope variables: `var`

The variable statement creates and initializes variables inside the function scope: `var myVar, myVar2 = 'Init'`. By default a declared yet not initialized variable has `undefined` value.
Plain and simple, developers use this statement from first JavaScript versions:

Try in JS Bin

```
// Declare num variable
var num;
console.log(num); // => undefined
// Declare and initialize str variable
```

```
var str = 'Hello World!';
console.log(str); // => 'Hello World!'
```

Hoisting and `var`

Variables declared with `var` are hoisted to the top of the enclosing function scope. If the variable is accessed before declaration, it evaluates to `undefined`.

Suppose `myVariable` is accessed before declaration with `var`. In this situation the declaration is **moved to the top** of `double()` function scope and the variable is assigned with `undefined`:

Try in JS Bin

```
function double(num) {
  console.log(myVariable); // => undefined
  var myVariable;
  return num * 2;
}
double(3); // => 6
```

JavaScript will move the declaration `var myVariable` to the top of `double()` scope and interpret the code this way:

Try in JS Bin

```
function double(num) {
  var myVariable;           // moved to the top
  console.log(myVariable); // => undefined
  return num * 2;
}
double(3); // => 6
```

The `var` syntax allows not only to declare, but right away to assign an initial value: `var str = 'initial value'`. When the variable is hoisted, the declaration is moved to the top, but the initial value assignment **remains** in place:

Try in JS Bin

```
function sum(a, b) {
  console.log(myString); // => undefined
  var myString = 'Hello World';
  console.log(myString); // => 'Hello World'
  return a + b;
}
sum(16, 10); // => 26
```

`var myString` is hoisted to the top of the scope, however the initial value assignment `myString = 'Hello World'` is not affected. The above code is equivalent to the following:

Try in JS Bin

```
function sum(a, b) {
  var myString;              // moved to the top
  console.log(myString);     // => undefined
  myString = 'Hello World'; // remains
  console.log(myString);     // => 'Hello World'
  return a + b;
}
sum(16, 10); // => 26
```

3. Block scope variables: `let`

The let statement creates and initializes variables inside the block scope: `let myVar, myVar2 = 'Init'`. By default a declared yet not initialized variable has `undefined` value.

`let` is a great addition introduced by ECMAScript 6, which allows to keep the code modular and encapsulated on a block statement level:

Try in JS Bin

```
if (true) {
  // Declare name block variable
  let month;
  console.log(month); // => undefined
  // Declare and initialize year block variable
  let year = 1994;
  console.log(year); // => 1994
```

```
}
// name and year or not accessible here, outside the block
console.log(year); // ReferenceError: year is not defined
```

Hoisting and `let`

`let` variables are registered at the top of the block. But when the variable is accessed before declaration, JavaScript throws an error: `ReferenceError: <variable> is not defined`. From the declaration statement up to the beginning of the block the variable is in a *temporal dead zone* and cannot be accessed.

Let's follow an example:

Try in JS Bin

```
function isTruthy(value) {
  var myVariable = 'Value 1';
  if (value) {
    /**
     * temporal dead zone for myVariable
     */
    // Throws ReferenceError: myVariable is not defined
    console.log(myVariable);
    let myVariable = 'Value 2';
    // end of temporary dead zone for myVariable
    console.log(myVariable); // => 'Value 2'
    return true;
```

```
  }
  return false;
}
isTruthy(1); // => true
```

`myVariable` is in a temporal dead zone from `let myVariable` line up to the top of the block `if (value) {...}`. If trying to access the variable in this zone, JavaScript throws a `ReferenceError`.

An interesting question appears: is really `myVariable` **hoisted** up to the beginning of the block, or maybe is just **not defined** in the temporal dead zone (before declaration)? The exception `ReferenceError` is thrown also when a variable is not defined at all.
If you take a look at the beginning of the function block, `var myVariable = 'Value 1'` is declaring a variable for the entire function scope. In the block `if (value) {...}`, if `let` variables would not cover the outer scope variables, then in the temporal dead zone `myVariable` would have the value `'Value 1'`, which does not happen. So block variables are *rough hoisted*.

In an exact description, when the engine encounters a block with `let` statement, first the variable is *declared* at the top of the block. At *declared* state the variable still cannot be used, but it covers the outer scope variable with the same name. Later when `let myVar` line is passed, the variable is in *initialized* state and can be used.
Check also this explanation.

`let` expansion in the entire block protects variables from modification by outer scopes, even before declaration. Generate reference errors when accessing a `let` variables in temporary dead zone ensures better coding practice: first declare - then use.
Both these restrictions are an effective approach to write better JavaScript in terms of encapsulation and code flow. This is a result of lessons based on `var` usage, where accessing the variable before declaration is a source of misunderstanding.

4. Constants: `const`

The constant statement creates and initializes constants inside the block scope: `const MY_CONST = 'Value', MY_CONST2 = 'Value 2'`. Take a look at this sample:

Try in JS Bin

```
const COLOR = 'red';
console.log(COLOR); // => 'red'
const ONE = 1, HALF = 0.5;
console.log(ONE);   // => 1
console.log(HALF);  // => 0.5
```

When a constant is defined, it must be initialized with a value in the same `const` statement. After declaration and initialization, the value of a constant cannot be modified:

Try in JS Bin

```
const PI = 3.14;
console.log(PI); // => 3.14
PI = 2.14; // TypeError: Assignment to constant variable
```

Hoisting and `const`

Constants `const` are registered at the top of the block.
The constants cannot be accessed before declaration because of the *temporal dead zone*. When accessed before declaration, JavaScript throws an error: `ReferenceError: <constant> is not defined`.

`const` hoisting has the same behavior as the variables declared with `let` statement (see hoisting and `let`).

Let's define a constant in a function `double()`:

Try in JS Bin

```
function double(number) {
   // temporal dead zone for TWO constant
   console.log(TWO); // ReferenceError: TWO is not defined
   const TWO = 2;
   // end of temporal dead zone
   return number * TWO;
```

```
}
double(5); // => 10
```

If `TWO` is used before the declaration, JavaScript throws an error `ReferenceError: TWO is not defined`. So the constants should be first declared and initialized, and later accessed.

5. Function declarations

The function declaration defines a function with the provided name and parameters. An example of function declaration:

Try in JS Bin

```
function isOdd(number) {
   return number % 2 === 1;
}
isOdd(5); // => true
```

The code `function isOdd(number) {...}` is a declaration that defines a function. `isOdd()` verifies if a number is odd.

Hoisting and function declaration

Hoisting in a function declaration allows to use the function anywhere in the enclosing scope, even before the declaration. In other words, the function can be called from any place of the current or inner scopes (no `undefined` values, temporal dead zones or reference errors).

This hoisting behavior is flexible, because you can first *use* the function and only later *declare* it. Or apply the classic scenario: first *declare* and then *use*. As you wish.

The following code from the start invokes a function, and after defines it:

Try in JS Bin

```
// Call the hoisted function
equal(1, '1'); // => false
// Function declaration
function equal(value1, value2) {
   return value1 === value2;
}
```

The code works nice because `equal()` is created by a function declaration and hoisted to the top of the scope.

Notice the **difference** between a **function declaration** `function <name>() {...}` and a **function expression** `var <name> = function() {...}`. Both are used to create functions,

however have different hoisting mechanisms.
The following sample demonstrates the distinction:

Try in JS Bin

```
// Call the hoisted function
addition(4, 7); // => 11
// The variable is hoisted, but is undefined
substraction(10, 7); // TypeError: substraction is not a function
// Function declaration
function addition(num1, num2) {
   return num1 + num2;
}
// Function expression
var substraction = function (num1, num2) {
  return num1 - num2;
};
```

`addition` is hoisted entirely and can be called before the declaration.
However `substraction` is declared using a variable statement (see 2.) and is hoisted too, but has an `undefined` value when invoked. This scenario throws an error: `TypeError: substraction is not a function`.

6. Class declarations

The class declaration defines a constructor function with the provided name and methods. Classes are a great addition introduced by ECMAScript 6.
Classes are built on top of the JavaScript prototypal inheritance and have some additional goodies like `super` (to access the parent class), `static` (to define static methods), `extends` (to define a child class) and more.

Take a look how to declare a class and instantiate an object:

Try in JS Bin

```
class Point {
    constructor(x, y) {
      this.x = x;
      this.y = y;
    }
    move(dX, dY) {
      this.x += dX;
      this.y += dY;
    }
}
// Create an instance
var origin = new Point(0, 0);
// Call a method
origin.move(50, 100);
```

Hoisting and `class`

The class variables are registered at the beginning of the block scope. But if you try to access the class before the definition, JavaScript throws `ReferenceError: <name> is not defined`. So the correct approach is first to *declare* the class and later *use* it to instantiate objects.

Hoisting in class declarations is similar to variables declared with `let` statement (see 3.).

Let's see what happens if a class is instantiated before declaration:

Try in JS Bin

```
// Use the Company class
// Throws ReferenceError: Company is not defined
var apple = new Company('Apple');
// Class declaration
class Company {
  constructor(name) {
    this.name = name;
  }
}
// Use correctly the Company class after declaration
var microsoft = new Company('Microsoft');
```

As expected, executing `new Company('Apple')` before the class definition throws `ReferenceError`. This is nice, because JavaScript suggests to use a good approach to first

declare something and then make use of it.

Classes can be created using a class expression, which involves variable declaration statements (with `var`, `let` or `const`). Let's see the following scenario:

Try in JS Bin

```
// Use the Sqaure class
console.log(typeof Square);   // => 'undefined'
//Throws TypeError: Square is not a constructor
var mySquare = new Square(10);
// Class declaration using variable statement
var Square = class {
  constructor(sideLength) {
    this.sideLength = sideLength;
  }
  getArea() {
    return Math.pow(this.sideLength, 2);
  }
};
// Use correctly the Square class after declaration
var otherSquare = new Square(5);
```

The class is declared with a variable statement `var Square = class {...}`. The variable `Square` is hoisted to the top of the scope, but has an `undefined` value until the class declaration line. So the execution of `var mySquare = new Square(10)` before class

declaration tries to invoke an `undefined` as a constructor and JavaScript throws `TypeError: Square is not a constructor`.

7. Final thoughts

As seen in the explanations, hoisting in JavaScript has many forms. Even if you know exactly how it works, the general advice is to code variables in a sequence of *declare* > *initialize* > *use*. ECMAScript 6 certainly suggests this approach by the way hoisting is implemented for `let`, `const` and `class`. This will save you from *unexpected* variable appearances, `undefined` and `ReferenceError`.

As an exception, sometimes functions can be invoked before the definition: an effect of function declaration hoisting. It's useful in cases when developer needs to read quickly how functions are invoked at the top of the source file, without the necessity to scroll down and read the details about function implementation.
For example, see here how this approach increases the readability of Angular controllers.

I hope you enjoyed the reading, so do not hesitate to it. See you in my next post :).

P.S. You might also want to check out:
Gentle explanation of 'this' keyword in JavaScript

The legend of JavaScript equality operator

JavaScript addition operator in details

javascript function variable hoisting ecmascript-2015

Dmitri Pavlutin

Web & mobile iOS developer. Swift and JavaScript languages fan. Coding, blogging, learning, open sourcing, solving problems - in a cycle is my routine.
About me

When 'not' to use arrow functions

It is a pleasure to see the evolution of the programming language···

Gentle explanation of 'this' keyword in JavaScript

Off topic preface I recently released my first open source JavaScript library:···