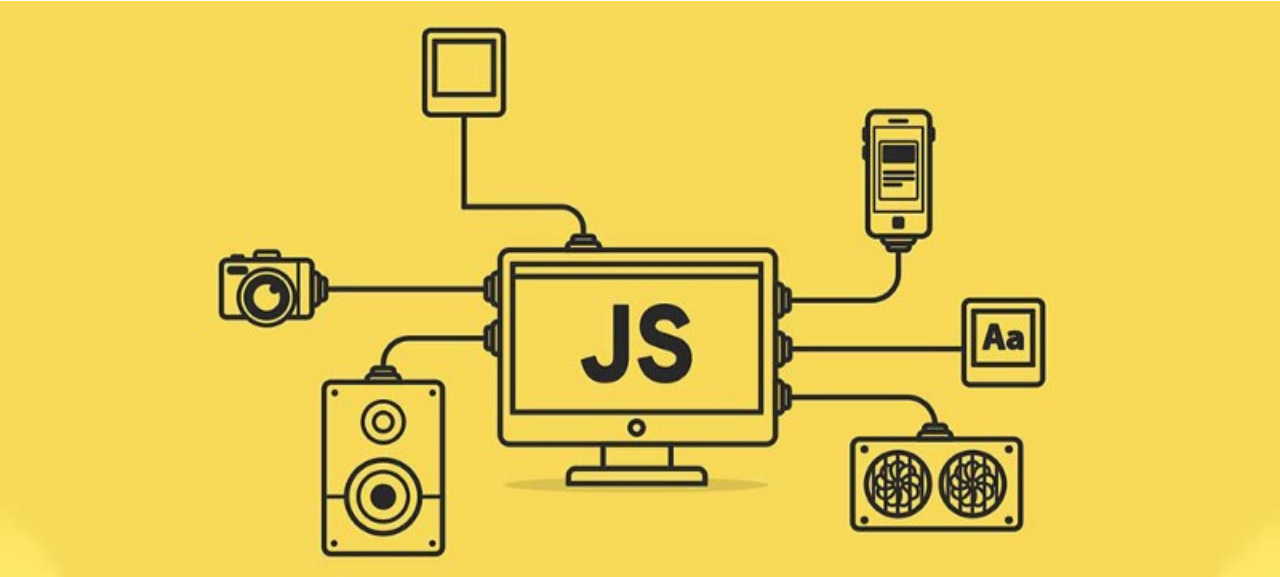
#001 JavaScript with Dimos: Variables and Constants

Hey there I am Dimos, a senior software engineer and software architect at arconsis. From time to time, I publish articles into the series "JavaScript with Dimos", where I am trying to analyze a JavaScript topic and to focus on the core concepts of JavaScript.



In the chapter "JavaScript's Grammar" we will take a look at JavaScript's basic grammar, variable declarations and data types.

Chapter's Posts Roadmap

- Variables and Constants (This article)
- Data Types

Today, in the first article of this chapter, we will explore JavaScript variables and constants, how to initialize and use them and their differences.

JavaScript Variables

A variable is just a container to **store** data and **hold** its value. For example we can use a variable to store a number, or a string:

```
var str = 'Hello!';
```

In above example, str is a variable we use to store a text.

Variable’s Declaration

It is essential to know how to declare variables in every programming language. The var, let, and const keywords are used to **declare** variables in JavaScript. There are some significant differences among them which we will analyze later in this tutorial.

```
var x = 1;
let y = 2;
const z = 3;
```

Variable’s Initialization

We can use the **assignment** operator = in order to initialize a variable and assign it a value. For example:

```
var num = 3;
let str = 'Hello!';
const isValid = true;
let str2;
str2 = 'Hey!';
```

In the above example, num was initialized with 3 as the initial value. As we can see, we can declare and initialize a variable at the same time (let str = 'Hello!';) or we can first declare it (let str2;) and then initialize it str2 = 'Hey!';.

JavaScript supports **multiple** variable declaration in a single sentence just like below:

```
let x = 1, y = 2, z = 3;
```

Variable’s Value Re-assignment

In JavaScript, it is possible to **mutate** the variable’s value, like below:

```
let x = 1;
console.log(x); // output 1
x = 2;
console.log(x); // output: 2
```

```
var y = 'Hello!';
console.log(y); // output: Hello!
y = 'Hey!';
console.log(y); // output: Hey!
```

JavaScript Constants

Similar to variables, we can declare **constants** in JavaScript. We can use const statement (was introduced in the ES6 version) to create constants. For example:

```
const x = 1;
```

Constants are similar to variables, apart from:

- declaration and initialization must take place at **same time**.
- **cannot** be re-assigned after initialization.

For example:

```
const x;  // Uncaught SyntaxError: missing = in const declaration
x = 1;
console.log(x);
```

```
const x = 1;
x = 3;  // Uncaught TypeError: invalid assignment to const 'x'
console.log(x);
```

> Be careful: Although we **cannot** mutate a constant (applies only to **primitive** values) we can still **mutate** (update, add, or remove) properties of an **object** declared by const. This is possible because the object declared by const is still pointing to the same object, even though its content has been changed.

```
const vehicle = { type : 'car'};
console.log(vehicle.type);  // output: "car"

vehicle.type = 'bike';
console.log(vehicle.type);  // output: "bike"
```

var statement

The var statement defines a **function-scoped** or **globally-scoped** variable (scopes will be discussed in next article). We can assign a value optionally, on variable declared by var. It does **not** support block-level scope, which means that a variable declared inside {…} (for example inside for or if statement) **cannot** be accessed outside the block, which potentially can lead to bugs.

```
var num = 1;
```

```
if (num === 1) {
   var num = 2;
   console.log(x); // output: 2
}

console.log(num); // output: 2
```

a) var is function-scoped

Function scope means that variables defined inside a function **cannot** be accessed from anywhere **outside** the function. For example:

```
function foo() {
   var x = 1;
   function baz() {
      var y = 2;
      console.log(x); // output: 1 (baz() can see outer scope)
      console.log(y); // output: 2
   }
   baz();
   console.log(x); // output: 1 (as x is in same function scope)
   console.log(y); // Uncaught ReferenceError: y is not defined
}

foo();
```

Another example:

```
var animal = 'dog';

function foo() {
   animal = 'cat';
   console.log(animal); // output: "cat"
}

console.log(animal); // output: "dog"
foo();
console.log(animal); // output:"cat"

// output: "dog", "cat", "cat"
```

b) var gets hoisted

The variables declared by var are processed before any code is executed in a process known as “**hoisting**” with undefined as initial value. If we declare a variable anywhere in the code using var, is the same as declaring it at the top. Hence a var can appear to be used **before** it’s declared, which is called

“**hoisting**”**.** In other words, the term “**hoisting**” means that all var and function declarations are “pulled” to the top of the code when the code is run.

```
animal = 'dog';
var animal;

// ...is equivalent (hoisting):

var animal;
animal = 'dog';

console.log(foo); // output: undefined
var foo = 1;
console.log(foo); // output: 1
```

Or another example:

```
function foo() {
  console.log(baz); // output: undefined
  var baz = 3;
  console.log(baz); // output: 3
}

// ...is equivalent (hoisting):

function foo() {
  var baz;
  console.log(baz); // output: undefined
  bar = 3;
  console.log(baz); // output: 3
}
```

> **Recommendation**: It’s best to declare variables always at the top of their scope, to be clear which is their scope.

c) var can be re-declared

Variables declared with var can be *re-declared* with a new value. For example:

```
var x = 1;
console.log(x); // output: 1
var x = 3;
console.log(x); // output: 3
```

d) var can be re-assigned

Variables declared by var can be *re-assigned* to a new value. For example:

```
var x = 1;
console.log(x); // output: 1
x = 3;
console.log(x); // output: 3
```

let statement

The let statement was introduced in the ES6 version. The let and the var statements have some similarities but their main difference is that let declares a **block-scoped** variable and although it can be reassigned it **cannot** be **re-declared**. Also compared to var statements, a variable declared using let will **not** be **hoisted**.

a) let is block-scoped

Unlike the var keyword, which defines a global or function-scoped variable, the let keyword allows us to define variables that are **block-scoped**. This means that variables are **limited** to the scope of the **block** statement or **expression** on which it is used, as well as in any nested **sub-blocks**.

```
let x = 3;

if (x === 3) {
   let x = 6;
   console.log(x); // output: 6
}

console.log(x); // output: 3
```

Another example:

```
function foo() {
   let x = 3;
   function baz() {
      let x = 6;  // different variable
      console.log(x);  // output: 6
   }
   baz();
   console.log(x);  // output: 3
}

foo();
console.log(x); // output: ReferenceError: x is not defined (defined in function scope, cannot be accessed outer)
```

b) let cannot be hoisted

The variables declared by let statement, **cannot** be hoisted and a ReferenceError is expected to be thrown, compared to variables declared by var. For example:

```
console.log(x); // Uncaught ReferenceError: can't access lexical declaration 'x' before initialization
let x = 1;
```

c) let cannot be re-declared

If we try to re-declare the same variable by let within the same function or same block, then a SyntaxError is expected to be thrown.For example:

```
if (true) {
   let x = 1;
   let x = 3; // Uncaught SyntaxError: redeclaration of let x
}
```

Or another example:

```
function foo() {
   let x = 3;
   let x = 6; // Uncaught SyntaxError: redeclaration of let x
}
foo();
```

Also if we combine var and let statements, then it is possible to get a SyntaxError, because var is hoisted to the top of the block. As we can see below, the variable declared by var , will introduce re-declaration of x , as is not block-scoped and will be “pulled” on top of the code because of the “hoisting”.

```
let x = 3;
{
   var x = 5; // Uncaught SyntaxError: redeclaration of let x, Previously declared
}
```

d) let can be re-assigned

Variables declared by let keyword can be re-assigned with a new value. For example:

```
let x = 1;
console.log(x); // output: 1
```

```
x = 3;
console.log(x); // output: 3
```

const statement

The const keyword and let keyword have a lot of similarities, as both are **blocked** scoped, **cannot** be hoisted and **cannot** be **re-declared**. The only difference is that variables declared by const **cannot** be **reassigned**. Hence, every const variable must be initialized with a value at the time of declaration.

a) const is block-scoped

The const statement allows us to define variables that are block-scoped. This means that variables are **limited** to the scope of the **block** statement or **expression** on which it is used, as well as in any contained **sub-blocks**.

```
const x = 3;
if (x === 3) {
   const x = 6;
   console.log(x); // output: 6
}
console.log(x); // output: 3
```

Or another example:

```
function foo() {
   const x = 3;
   function baz() {
      const x = 6;  // different variable
      console.log(x);  // output: 6
   }
   baz();
   console.log(x);  // output: 3
}

foo();
console.log(x); // output: ReferenceError: x is not defined (defined in function scope, cannot be accessed outer)
```

b) const cannot be hoisted

The variables declared by const statement, **cannot** be hoisted and a ReferenceError is expected to be thrown, compared to variables declared by var. For example:

```
console.log(x); // Uncaught ReferenceError: can't access lexical declaration 'x' before initialization

const x = 1;
```

c) const cannot be re-declared

If we try to re-declare the same variable by const within the same function or same block , then a SyntaxError is expected to be thrown. For example:

```
if (true) {
   const x = 1;
   const x = 3; // Uncaught SyntaxError: redeclaration of let x
}
```

Or another example:

```
function foo() {
   const x = 3;
   const x = 6; // Uncaught SyntaxError: redeclaration of let x
}
foo();
```

Also if we combine var and const statements, then it is possible to get a SyntaxError, because var is hoisted to the top of the block. As we can see below, the variable declared by var , will introduce re-declaration of x , as is not block-scoped and will be “pulled” on top of the code because of the “hoisting”.

```
const x = 3;
{
   var x = 5; // Uncaught SyntaxError: redeclaration of let x, Previously declared
}
```

d) const cannot be re-assigned

Variables declared by const keyword **cannot** be re-assigned with a new value and a TypeError is expected to be thrown. For example:

```
const x = 1;
console.log(x); // output: 1
x = 3;
console.log(x); // output: TypeError: invalid assignment to const 'x'
```

> Note: If we know that the variable’s value will not change, it’s preferable to use const instead of var or let.

Those are all we need to know about **variables** and **constants** declaration in JavaScript and their differences. Meanwhile, please feel free to post any questions, comments, or feedback in the comment section below.

Follow on Twitter here!

Follow on Github here!

See you until next time!

References

1. https://www.programiz.com/javascript/variables-constants
2. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var
3. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let
4. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const
5. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Grammar_and_Types
6. https://github.com/getify/You-Dont-Know-JS

By Dimosthenis Botsaris on May 12, 2022.

Canonical link

Exported from Medium on May 12, 2022.