1

Variables: Declaration, Definition and Type

Variables are made to receive values directly or through evaluated expressions compounding values and operators or results of function calls: the value is stored internally and the variable references this storage, and takes its value and its type.

There are two kinds of values:

– primitive values: numbers or strings in literal notation, "built-in" values such as `true`, `false`, `NaN`, `infinity`, `null`, `undefined`, etc.;

– objects, including functions, and arrays, which are "containers", and, as such, their value is the address of the container.

NOTE.– The content of a container can be modified, while the address remains unchanged (this is important for understanding the "`const`" declaration below).

To fetch the value of a variable, it must have been identified in the lexical phase: which means to be "declared". Often, JavaScript code starts with declaration instructions such as:

```
var tableauCandidats = [];
var n = 0; // ... ...
```

We will show why it is highly preferable to write:

```
const tableauCandidats = [];
let n = 0; // ... ...
```

Let us look back at the two steps in the interpretation of JavaScript (simplified):

– Lexical-time: It deals with the lexical analysis of the code. The names of the declared functions and variables are recorded in a tree structure (lexical tree). Declarations are moved up to the beginning of their block of code: this is named the "hoisting". Functions are hoisted first, for they define the nodes of the structure, then variables are attached to appropriate nodes.

– Run-time: The script engine reads the instructions again and runs them one by one, with the help of the tree structure. Expressions (right-hand side of assignment instructions) are evaluated, and values are assigned to variables (on the left-hand side) with the associated type (dynamic typing).

Let us detail this double mechanism: declaration of the variables, initialization to `undefined` at *lexical-time* and definition of the variables at *run-time*.

1.1. Declarations of functions and variables

For a better understanding of the declaration mechanism in JavaScript, we must first learn what the "scope" of a variable is: the function declarations determine the overall logics of the notion of scope.

1.1.1. *The different declaration keywords*

1.1.1.1. *Function declaration*

The keyword comes first in the instruction line; the syntax is:

```
function name (list, of, parameters) { block_of_code }
```

At lexical time, the name is stored in the lexical tree as a new function node. The list of parameters, and all variables and functions declared within the block, are added to this node. New declared functions open new nodes as subnodes: this is a recursive process (depth-first analysis). We will detail this in Chapter 6, section 6.4.1.1.

Therefore, every function determines a "function scope". At the beginning of the lexical analysis, the engine uses a "root node" that is named the "global scope".

NOTES.–

1) A variable that does not appear in the lexical tree (i.e., never declared) cannot be assigned to another variable; for its evaluation, it is impossible: a `Reference Error` is triggered.

2) An attempt to assign a (evaluable) value to a never declared variable, for instance, `x = 1` with `x` absent from the lexical tree, is not an error. Therefore, the variable is added to the global scope. This is bad practice, an "anti-pattern" (see case [f4]).

1.1.1.2. *`var` declaration*

Before 2015, the only way to declare a variable was the old-fashioned declaration keyword, which can be used in one of these forms:

```
var x;                                  // simple declaration
var x = [[val]];                        // declaration with definition
var x = [[val]], y = [[val]], z;        // multiple declarations
```

The declaration is hoisted at the beginning of the function scope or the global scope. The variable, if not explicitly defined with a value, receives the value undefined. That value is hoisted together with the reference.

NOTE.– If JavaScript is embedded in a browser, the "global scope" is the `window` object:

```
var a; // equivalent: 'window.a' (a as a property of window)
function f(){var a;} // this 'a' is different from 'window.a'
```

1.1.1.3. *`let` declaration*

The keyword `let` acts as `var`, and moreover limits the scope of the variable to the context of a block: for instance, a conditional instruction block, a function block or the global scope.

There is another difference: no "hoisting" is applied to the variable, hence there is a gap between the lexical existence (whole block) and the existence of the reference (declaration line). This means that, during that gap, any attempt to fetch that reference will trigger an error:

```
{ /* new block */
  console.log(x);
```

```
// ReferenceError: can't access lexical declaration x before initialization
  let x = 4;
}
```

And there is an additional constraint: it is forbidden to redeclare the same name in the same block:

```
let x = 1; let x = 2;  // SyntaxError: redeclaration of let x
let x = 1; x = 2;      // the redefinition is allowed
```

These constraints provide a better protection against unwilling modifications of the lexical context.

1.1.1.4. *const declaration*

In this, the keyword behaves like `let` with two additional constraints:

– declaration AND definition must be done in the same instruction, and any redeclaration is forbidden:

```
const Euler; // SyntaxError: missing = in const declaration
```

– redefinition is forbidden:

```
const Pi = 3.14; // the value of Pi est defined only once
Pi = 3.1; // TypeError: invalid assignment to const `PI'
```

– redeclaration with the same or a different keyword is equally forbidden:

```
const pi = 3; let pi = 3; // SyntaxError: redeclaration of const
function y(){}
let y; // SyntaxError: redeclaration of function y
```

Note about `const`: A variable already defined cannot be redefined, which means that if the value is an object, you will always use the same object, as a container, but its properties can be modified as often as you need.

Therefore, the use of const is highly recommended for objects, arrays and function's expressions.

Note about the three keywords: `var` is the most "permissive", hence the most error prone; `const` is the most constraining, hence detects the greatest number of coding inconsistencies, right at the lexical stage:

The recommendation is to privilege const, unless you know that the variable is used temporarily, and will evolve soon. For instance, an index, a cumulative value, a boolean control, etc.

1.1.2. *Lexical scope and definition of a variable according to declaration mode: `var`, `let`, `const`*

Let us present some examples to illustrate the differences between pre-ES6 and post-ES6 situations, depending on four different cases. In all the following cases, we assume that the variable `x` is never declared elsewhere in the global scope.

1.1.2.1. *Situation pre-ES6*

Here are four functions, corresponding to four cases. In Tables 1.1 and 1.2, we fetch the type and value of `x`, within or outside the function, and before and after the declaration instruction. The use of `var` shows how "permissive" and risky it is.

Pre-ES6. Four cases for a variable in a function scope (or not)

```
        // f1: no declaration, no definition of 'x'
function f1() { /* no occurrence of x in function */ }
        // f2: declaration of 'x' but no definition
function f2() { /* local before */ var x;     /* local after */ }
        // f3: declaration an definition in the same instruction,
function f3() { /* local before */ var x = 1; /* local after */ }
        // f4: assignation of a value to 'x' without declaration.
function f4() { /* local before */ x = 1;     /* local after */ }
```

f1 Local	{ type: undefined}, val -> ***ReferenceError: x is not defined***
global before call	{ type: undefined}, val -> ***ReferenceError: x is not defined***
global after	{ type: undefined}, val -> ***ReferenceError: x is not defined***
f2 local, before var	{ type: undefined, val: undefined};
after var	{ type: undefined, val: undefined };
global after	{ type: undefined}, val -> ***ReferenceError: x is not defined***

f3 local, before var	{ type: undefined, val: undefined },
after var	{ type: number, val: 1 };
global after	{ type: undefined}, val -> ***ReferenceError: x is not defined***
f4 local, before =	{ type: undefined}, val -> ***ReferenceError: x is not defined***
after =	{ type: number, val: 1 };
global after	{ type: number, val: 1 }; // !!! beware !!!

Table 1.1. *Lexical scope and value of a variable (four cases), using var declaration*

COMMENTS.–

[f1]: any attempt to fetch the variable always results in type `undefined` (the operator `typeof` never throws an error) and a "Reference Error".

[f2]: type and value are "undefined" inside, no reference outside;

[f3]: type and value are updated: "undefined" -> "number", no reference outside;

[f4]: *the most weird and dangerous*: at run-time, the instruction `x = 1` does not find any lexical reference for `x`, and the JavaScript engine creates one in the global space. Silently! Which may result in later troubles.

NOTE.– A very frequent and unwilling cause for [f4] is when using loop index: `for(i = 0; i <length; i++) { /* code */ }` // DANGER ! Use `'let i'` or -better-, try to avoid the loop (Chapter 5).

1.1.2.2. *Situation post-ES6*

Now, let us use `let` or `const`, wherever `var` was used. There is no difference where `var` was not used, hence, this is not repeated here, and the case f4 is not changed either.

Post-ES6. Different cases for a variable in a function scope (or not)

```
function f5() { /*local before*/ let x;      /*local after*/ }
function f6() { /*local before*/ let x = 1;  /*local after*/ }
function f7() { /*local before*/ const x = "Man"; /*local after*/ }
/* f8 would be strictly identical to f4 */
```

f5	local before	{ type: undefined} -> ***ReferenceError: can't access lexical declaration before initialization***
	after	{ type: undefined, val: undefined};
f6	local before	{ type: undefined} -> ***ReferenceError: can't access lexical declaration before initialization***
	after	{ type: undefined, val: 1};
f7	local before	{ type: undefined} -> ***ReferenceError: can't access lexical declaration before initialization***
	after	{ type: string, val: "Man" }; mandatory
all	cases: global	{ type: undefined} -> ***ReferenceError: x is not defined***

Table 1.2. *Lexical scope and value of a variable using let or const declaration*

1.1.3. *Comments (important improvements carried over by ES6)*

Cases [f5] and [f6] are the equivalent of [f2] and [f3]: The difference with `let` is that we cannot use the variable before its declaration. The text of the error tells us that the problem resides *within the block*; if outside, the message would be: ***ReferenceError: x is not defined***. This is very useful for tracking bugs.

When using `const`, the only possible case is [f7], equivalent of [f3]: if you forget to assign a value, the error is: ***SyntaxError: missing = in const declaration***.

1.1.4. *General conclusion about the variable declarations in JavaScript*

Considering the four possible cases with declarations:

– `const` is the most constraining, leading to less risky situations: limited lexical scope, no redeclaration, no redefinition and clearer messages. Moreover, the declaration and the definition are necessarily done by a single instruction line, which results in (a) more readable code and (b) immutable type. An immutable type is better for helping the script engine to optimize its bytecode, resulting in better performance.

– `var` is the most permissive, hence the most risky: unwilling redeclarations are a frequent cause of silent error. In this, a keyword exists for legacy reasons; you must ban it from your code.

– `let` should be used in the cases where you intend to reuse the same variable name: incrementing a number, or modifying a string, or a boolean constant. For referencing containers (Objects, Arrays, Functions), use `const` (see the following chapters). For strings or numbers that are not supposed to change, use `const` as well.

– absence of declaration: when a variable appears, for the first time, on the left-hand side of an assignment instruction, without a declaration keyword, an implicit declaration is done. It is a programming design mistake, but does not cause a problem for JavaScript. See next sub-section "implicit addition".

Figure 1.1 shows the mechanism of the interpretation of the script, when using the recommended keywords `const` and `let`, for `x` and `y`, and with a forgotten keyword before `z`.

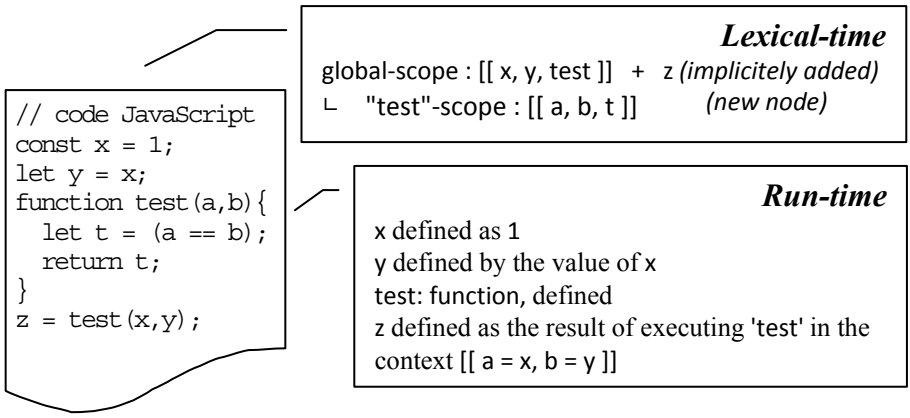

Figure 1.1. *The two phases of the interpretation mechanism*

NOTE 1.– One advantage of using `const`, if you forget to declare a variable:

```
const Pi = 3.14;      // global: can't be redefined
function calcul(x){
   Pi = 3.1416;       // a future new global at run-time
   return 2*Pi * x;
}
calcul(2); //TypeError: invalid assignment to const
```

NOTE 2.– If a variable cannot be declared with `const` because you must modify it, you can embed this variable as a property of an object (see

Chapter 4) and declare that object with `const`: you will benefit from the advantage of Note 1:

```
const xObj = { x = 3 }; // global: can't be redefine
function calcul(x){
   xObj.x = x; // you can modify the property xObj, but
   xObj = x; // will provoque an error at run-time
}
calcul(2); //TypeError: invalid assignment to const
```

1.1.4.1. *The implicit addition of a variable in the "global scope" (at run-time)*

We have seen that an instruction `x = [[val]]` may result in an implicit declaration in global scope, if the variable is not present in the lexical tree.

The instruction is ignored at lexical time, and no "hoisting" is made, but at run-time, the name `x`, on the left-hand side of a valid assignment instruction, is not found in the lexical tree: hence `x` is added to the global scope.

Note that an instruction x = x+1 that would have thrown a "ReferenceError: x is not defined" for the evaluation of the right-hand side is impossible.

1.1.4.2. *Wrapping it up*

The reasons to ban `var` are as follows:

Var	Let

Redeclaration is possible: YES
`var k = 10;`
`var k = "dix"; // allowed`
	Redeclaration: NO
	`let k = "dix";`
	`let k = 10; // Error!`

Limitated to a block: NO
`// i EXISTS before = undefined`
`for( var i = 0; i < 5; i++ ) {`
`  /* i ok in the loop */ }`
`// i EXISTS after = 5`
	Limitated to a block: YES
	`//Error! attempt using i before`
	`for( let i = 0; i < 5; i++ ) {`
	`  /* i ok in the loop */ }`
	`//Error! if using i after`

Hoisting: YES
`// n HOISTED = undefined`
`console.log( n ); // undefined`
`var n = 12; // = 12`
	Hoisting: NO
	`// n doesn't exist`
	`console.log( n ); // Error!`
	`let n = 12; //statement dropped`

Table 1.3. *The different behavior of var and let or const*

1.1.5. *Naming variables and functions: best practices*

1) Do not use “reserved words” (see Chapter 7) for it is forbidden. For instance, `function class(c){return "color:"+c;}` will throw a Syntax error because `class` is a reserved word.

2) Never use the same name in two declarations. Using `const` and `let` only protects you from doing so, but several function declarations will not fail: the last one prevails, which may cause damage elsewhere in the code, which is hard to debug.

3) Use good naming practices to facilitate reading of your code:

- avoid meaningless names, except for short-term, buffer-like variables in short blocks of code (typically less than 10 lines of code);

- use “camelCase” notation: it splits words, while avoiding the space (“camel case”) or dash (“camel-case”) which would be misinterpretated;

- use upper case initial letters only for functions that you intend to use as object constructor (e.g. `let d = new Date(););`

- a constant value can be named in full upper case: `const FRANC = 6.56;`

4) Limit the number of global variables to a minimum, possibly to 1:

REASON.– The JavaScript engine requires an environment, the Global Object. In the environment of the browser, the global object is `window`, an already very “crowdy” object. Every new variable AND function, created in the code, and which is not included in the block of code of a function, ends up in the global object:

```
/* code at the global level */
function f(){  /*local code*/
}
function g(){  /*local code*/
}
const obj1 = {"a", "b", "c"};
const tabInitial = [];
```

```
/* equivalent to */
window.f = function(){ ... };
window.g = function(){ ... };
window.obj1 = ...
window.tabInitial = ...
```

And the window becomes more and more crowded, which is a real performance issue. The solution includes all the code in a single function. This practice is called “the local function”.

1.2. Variable definition, initialization and typing in JavaScript

1.2.1. *Variables initialization and definition*

The `let`-declared variables are defined at run-time, when they appear on the left-hand side of an assignment. Between their declaration (`let`) and the assignment (=), their value is undefined.

```
let x, y;              // x and y lexically declared.
x = 1;          // x is assigned the value 1
y = x;          // y is assigned the value of x, hence 1
      // or, in a different order
let x, y;              // x and y lexically declared
y = x;          // y is assigned the value of x, hence 'undefined'
x = 1;          // x is assigned the value 1
```

The `const`-declared variables are declared and initialized just once, avoiding most of these subtleties: use `const` as often as possible.

If you cannot use `const`, at least use `let` in a combined declaration and definition instruction, and preferably at the beginning of the block: this will avoid the definition gap. And this is good for the script engine: the sooner it knows the type, the sooner the bytecode optimization can be applied.

1.2.2. *Types*

We can fetch the type of any variable, thanks to the operator `typeof`.

A function receives the type `"function"`. For any other variable, the value and the type are determined, at run-time, when the first assignment instruction is met for that variable. With a `const` declaration, definition and type come at the same time, which is the best situation for code robustness and performance considerations.

With `let`, we would rather combine declaration and definition:

```
let x; console.log(x +", "+ typeof x); // undefined, undefined
x = 4;                                 // only now: 4, number
let y = "4"; console.log(y +", "+ typeof y);    // 4, string
let v = (x>3); console.log(v +", "+ typeof v);  // true, boolean
```

In other cases that should be avoided, namely `var` and "implicitly defined" variables, the value and the type are set by default to `"undefined"`.

Actually this is a "*dynamic typing*", because variables always have a type, which can be `"undefined"`, and the initial type can be modified, except with `const`. Another reason for using `const` in priority.

Values in JavaScript are either "primitives" or objects:

– a *primitive* value has no property nor method (see note below): these are built-in values (`undefined`, `null`, `true`, `false`, `NaN`), numbers, or strings;

– an object is a collection of named values, couples: {name:value}.

There are six types in JavaScript:

– Type for the undefined variables: `"undefined"` (valeur `undefined`)

– Type: boolean (possible values: `true` or `false`)

– Type: number (e.g. `42`, `3.14159`, `0x0F`, `Infinity`, `NaN`)

– Type: string (e.g. `"JavaScript"` or `""` for the empty string)

– Type: function const f = function(){}; // typeof: "function"

– Type: object, for any object that is not a function, and includes `null`.

```
const obj = {"nom": "X", "age": 23};  // typeof obj -> object
```

There are many built-in objects in JavaScript, which all receive the type `"object"`: `Object`, `Array`, `String`, `Number`, `Boolean`, `Date`, `RegExp`, `Error`, `Math`, `JSON`. The exception is the object `Function`, whose type is `"function"`. There also exists the object `null` that makes it possible to initialize explicitly a variable that is unknown (and empty) and could presumably be set later.

NOTE.– There exist objects, `Number` and `String`, whose only role is to "bind" the corresponding types `"number"` and `"string"` in order to provide their values with appropriate methods. For similar reasons, there exist `Array`, `Function` and `Object`.

You should avoid creating such objects, in particular for primitive values, and rather use their literal notations which we will learn in each respective chapter.

1.2.3. *How to use the type "undefined" and the value undefined*

Never explicitly assign the value `undefined` to define a variable, though it is allowed. The reason for not using undefined, together with other previous recommendations, is:

a) a variable declared with `const` cannot be `undefined`;

b) a variable declared with `let` with an assignment, is not `undefined`.

Hence, a variable with an `undefined` value comes from an "implicit declaration", which is easier to determine, by checking:

```
if(typeof x === 'undefined'){ alert("declaration is missing"); }.
```

In situations where it is impossible to define a variable at the time of the declaration, we use:

```
let x = null;
```