JavaScript Vanilla

Les 5 bases de l'algorithmie, appliquées au JS

Sommaire

Clique sur le bouton ▲ pour revenir au sommaire.

1. Les variables

▲

Avant de débuter, qu'est-ce qu'une variable ? Et bien c'est un contenaire, dans lequel on peut stocker n'importe quel objet, pouvant être modifié (si ce n'est pas une constante). Mais dans le cas du JS, qu'est-ce qu'un objet ? Absolument tout. Que ce soit une chaîne de caractère, un nombre, un booléen, une liste, un dictionnaire... ou même une fonction, tout est un objet.

Assigniations des variables

Pour créer une variable, il faut l'instancier avec une des trois instructions : `var`, `let` ou `const`, avec cette syntaxe : `<inscruction> <clé> = <valeur>`

```
var variableGlobale = 'exemple';
let variableCourante = 'exemple';
const constanteGlobale = 'exemple';
```

Pour modifier une variable, il ne faut pas réécrire cette instruction :

```
let exemple = 'Variable Super Chouette'
console.log(exemple) // = Variable Super Chouette
exemple = 'Variable beaucoup moins chouette'
console.log(exemple) // Variable beaucoup moins chouette
```

Ces trois instructions n'existent pas pour rien, il y a bien évidement des différences : L'instruction `let` permet de déclarer une variable dont la portée est celle du bloc courant. L'instruction `var` correspond à une variable globale, qui ne peut être ré-assginée. La variable pourra cependant être modifiée, contrairement à une constante, avec l'instruction `const`, qui est immuable.

Exemple pour let :

```
let x = 1;

if (true) {
  let x = 2;
  console.log(x); // = 2
}

console.log(x); // = 1
```

`x` a été réassginée avec l'instruction `let`. Dans le bloc `if`, `x` correspond à une autre variable que celle en dehors, c'est pour cette raison qu'à la dernière ligne, `x` reste inchangée. Sans let :

```
let x = 1;

if (true) {
  x = 2;
  console.log(x); // = 2
}

console.log(x); // = 2
```

Dans ce cas, `x` n'a pas été redéfinis, donc il correspond à la même variable.

A contrario, une variable initiée avec l'instruction `var` ne peut pas être réassignée :

```
var x = 1;

if (true) {
  var x = 2; // ERREUR : La variable x a déjà été assignée.
  // Le reste du code ne va pas s'effectuer.
}
```

Pour `const`, il n'est ni possible de réassigner...

```
const x = 1;
const x = 1; // ERREUR : x a déjà été assignée.
}
```

... ni possible de modifier la variable.

```
const x = 1;
x = 2; // ERREUR : x est une constante.
}
```

Je déconseille fortement d'utiliser l'instruction `var`. Dans le cas du développemment web, l'algorithme ne s'execute pas de manière séquencielle : On ne sait pas à quelle moment l'utilisateur va faire telle ou telle action. Le code peut s'executer de manière un peu aléatoire, il est donc préférable d'utiliser des variables courantes (avec `let`).

Une variable ne doit être défini que par des caractères simple. Les lettres de a à z. Il est possible de mettre des majuscules, mais il est d'usage de commencer par une minuscule.

```
let pommeDeTerre;
```

Contre-exemple : Les composants en React ont des noms qui commencent par des majuscules.

a. Les chaînes de caratère

Une chaîne de caractères et simplement une suite de caractères, entourés de doubles quotes et de simple quote (`' '` ou `" "`).

```
let chaine1 = 'Exemple 1'
let chaine2 = "Exemple 2"
```

Que faire si je veux utiliser une apostrophe ou des parenthèses ? Il faut utiliser un contre-slash `\` :

```
let chaine = 'Je m\'appelle professeur'
console.log(chaine) // Je m'appelle professeur
```

Il est possible de concatener plusieurs chaines de caractères : (attention aux espaces)

```
let chaine1 = 'Exemple 1'
let chaine2 = "Exemple 2"
let chaine3 = chaine1 + ' ' + chaine2
console.log(chaine1 + chaine2 + chaine3 + "fin")
// = Exemple 1Exemple 2Exemple 1 Exemple 2fin
```

Il n'est pas contre évidement pas possible de multiplier, soustraire ou diviser une chaine de caractères.

b. Les chiffres

```
// Un entier
let chiffre = 15
// Un nombre à virgule
let chiffre2 = 12.3
// On peut faire des oppération directement dans la définition de la variable...
let chiffre3 = 10 + 5 + (1.3 * 10) - (14 / 2)
// Ou dans une modification
chiffre3 = 10 + 2
chiffre3 = chiffre3 + 2
// Il est possible de faire des opérations comme ceci :
chiffre3 += 10
chiffre3 -= 10
chiffre3 /= 10
chiffre3 *= 10
```

Il est possible de faire le même genre d'opération (uniquement l'addition) avec des chaînes de caractère :

```
```

```
let a = 'Bonjour'
a += ' à toi'
```

Il est aussi possible d'additionner un chiffre et une chaine de caractère :

```
let a = 'Hello' + 12 // Hello12
```

c. Les booléens

Il n'existe que deux valeurs possibles pour un booléen

```
let bool = true
let bool2 = false
```

Les opérateurs logiques (déjà vu en maths) : ==, === : égal !==, != : non égal && : et, les deux doivent être vrais || : ou, un des deux au minimum doit être vrai <, > : inférieur, supérieur <=, >= : inférieur ou égal, supérieur ou égal

```
let bool = true
let bool2 = false
let bool3 = 10 !== 12 // => vrai
let bool4 = bool && bool3 // => vrai
let bool5 = bool || bool2 // => vrai
let bool6 = (false && true) || (bool3 || (bool2 && true)) // => vrai
```

d. Les listes

Les listes sont des objets à indexage numérique, nommé plus communément array qui peuvent contenir tout type de donnée.

```
let liste = ['Oui', 14, true, [3, 'Non', false]]
// Pour accéder à un élement de la liste, il faut référencier son index en commençant par 0
console.log(liste[0]) // = Oui
// L'élement 3 est une liste, mais elle n'agit pas comme une extension de la liste, c'est un élement en lui même
console.log(liste[3]) // = [3, 'Non', false]
console.log(liste[3][1]) // = Non
```

e. Les dictionnaires

Les dictionnaires sont comme les listes, mais leur indexage est libre : ce peut être une chaîne de caractère, un chiffre, ou même un booléen.

```
// L'indexage peut être noté sans quotes quand la chaîne de caractère n'a pas de d'espace
let dict = {
    hello: 1,
    dsd: 'salut',
    2: 'skdf',
    true: "dfs",
    'salu t': {
        'coucou': 'salut'
    }
}
// Un élement peut être récupéré comme pour les listes
console.log(dict['hello']) // 1
// Attention, sans les quotes, on va ici chercher l'index qui est stocké dans la variable nommée hello
console.log(dict[hello]) // Erreur, hello n'est pas définie
// Il est donc possible de stocker un index dans une variable
let index = 'dsd'
console.log(dict[index]) // salut
```

On considère que chaque élement d'un dictionnaire est un variable héritée de cet objet.

```
// Pour le cas des chaines de caractère, il est possible de récupérer la variable de cette manière
console.log(dict.hello)
```

Je reparlerai des élements hérités quand je parlerais des fonctions.

f. Les données nulles.

```
let a = null
let b = undefined
let b2; // undefined
let c = NaN // Not a number
```

2. Les conditions

▲

`if`, `else` et `else if` sont les trois instructions de cette partie, respectivement en français : "si", "sinon", et "sinon si". Un bloc `if` ne va s'executer que **si** une **condition** booléenne est remplie. Une condition booléenne est simplement une valeur booléenne, comme on a pu le voir dans >cette partie<. On peut créer un bloc `else if` à la suite d'un bloc if, qui s'executera si le bloc `if` ne s'execute pas. Et à la suite d'un bloc `if` ou `else if`, il est possible d'inicier un bloc `else`, qui s'executera si aucune condition n'a été `true`.

```
if (/*condition 1*/) {
  // Code à executer si la condition 1 est vrai
} else if (/*condition 2*/) {
  // Code à executer si la condition 1 est fausse mais que la condition 2 est vrai
} else {
  // Code à executer si aucune des condition n'est vrai
}
```

Il est possible de mettre une infinité `else if`.

On peut aussi écrire une condition `if` sans pour autant créer de bloc entier, si on ne veut mettre qu'une seule expression :

```
let x = 12;
if (x < 13) x += 24;
console.log(x) // => 36
```

On peut appliquer cette logique à toute la chaine d'instruction :

```
if (false) console.log('là')
else if (true) console.log('ici')
else console.log('par ici')
// => ici
```

3. La lecture et l'écriture

▲

La lecture

La lecture est l'entrée utilisateur. Dans le cas du JavaScript Vanilla, la lecture se fait à proprement parlé dans le navigateur. Il n'est pas nécessaire de faire une partie entière sur ce point.

L'écriture

L'écriture est ce qui sort de l'algorithme. Il y a deux moyen d'écrire dans la console :

```
console.log('Bonjour')
process.stdout.write('Bonsoir')
```

On parle depuis le début de `console.log`, mais c'est intéressant de savoir qu'il existe un autre moyen d'écrire dans la console. Voici un lien pour comprendre les différences entre les deux : Difference between process.stdout.write and console.log in Node.js

Pour rester avec le plus courant, et celui que vous allez le plus utiliser, parlons un peu plus de `console.log()`. Il est possible de faire des opérations dans la fonction :

```
console.log(12+4) // 16
```

On peut mettre plusieurs arguments :

```
console.log('Hello', 'toi', [1, 4, 5]) // Hello toi [1, 4, 5]
```

Ils seront compris comme des élements à imprimer les un à la suite des autres dans la console, sans saut à la ligne (\n) mais avec un espace.

Voici d'autres méthodes de la console utiles à connaître : La méthode `console.count(<label>)`. Elle affiche le nombre de fois que la ligne a été appelée avec un label donné.

```
console.count('Testing')
console.count('Testing')
console.count('Testing')
// Résultat avec le label Testing
/*
Testing: 1
Testing: 2
Testing: 3
*/
```

La méthode `console.table(<liste|dictionnaire>)`. Affiche les données d'une liste ou d'un dictionnaire sous forme de tableau.

```
console.table([['Paul', 'Marc', 'Dominique'], [19, 67, 69]])
console.table({
    Marc:{
        age:39,
        profession:'Professeur',
        list:['p', 'a', 124],
        exemple:{
            one:1,
            two:2
        }
    },
    Dominique:{
        age:52,
        profession:'Technicien',
        list:['o', 'q', 19],
        exemple:{
            one:3,
            two:4
        }
    }
})
// Résultat
/*
┌─────────┬──────┬──────┬─────────────┐
│ (index) │  0   │  1   │      2      │
├─────────┼──────┼──────┼─────────────┤
│    0    │ 'Paul' │ 'Marc' │ 'Dominique' │
│    1    │  19  │  67  │     69      │
└─────────┴──────┴──────┴─────────────┘

┌─────────┬─────┬──────────────┬──────────────────┬──────────────────────┐
│ (index) │ age │  profession  │       list       │       exemple        │
├─────────┼─────┼──────────────┼──────────────────┼──────────────────────┤
│  Marc   │ 39  │ 'Professeur' │ [ 'p', 'a', 124 ] │ { one: 1, two: 2 } │
│ Dominique │ 52  │ 'Technicien' │ [ 'o', 'q', 19 ] │ { one: 3, two: 4 } │
└─────────┴─────┴──────────────┴──────────────────┴──────────────────────┘
```

```
*/
// Attention ! Les listes ou dictionnaires imbriquées ne s'affichent pas sous la forme d'un tableau
```

La méthode `console.group()` crée un nouveau groupe en ligne, correspondant à un nouveau niveau d'indentation. `console.groupEnd()` referme ce groupe. `console.groupCollapsed()` crée un groupe qiu se referme (avec un bouton pour ouvrir/fermer le bloc). Cette méthode ne fait rien dans le terminal de l'ordinateur, il ne fonctionne que dans le navigateur.

```
console.log('1')
console.group()
console.log('2')
console.groupCollapsed()
console.log('3')
console.groupEnd()
console.log('4')
console.groupEnd()
console.log('5')
// Résultat
/*
1
  2
    3
  4
5
*/
```

Et une dernière méthode, celle-ci utilisé pour connaitre l'efficatité du code. La méthode `console.time(<label>)` démarre un nouveau chronomètre, qui d'arrêtera avec la méthode `console.timeEnd(<label>)`.

```
console.time('Timeur')
let cpt = 13*32
cpt -= 12
cpt *= 13
```

```
cpt += 12
console.log(cpt)
console.timeEnd('Timeur')
// Résultat avec le label Timeur
/*
5264
Timeur: 13.44ms
*/
```

Pour voir toutes les méthodes inhérentes à la console, n'hésiter pas à consulter la documentation

4. Les boucles

▲

Il existe deux type de boucle : Les boucles Tant que, et les boucles Pour.

Petit exemple pour comprendre la différence : Si un politique dit "**Pour** 150 Millions de chomeurs, je vais créer 150 Millions d'emplois". La limite est bien définie, c'est comme une boucle qui va s'effectuer 150 Millions de fois, et qui va à chaque fois créer un emplois. La boucle à une fin. Si un politique dit "**Tant** qu'il y aura des chomeurs, je vais créer des emplois". La boucle peut s'effectuer à l'infini, car la limite n'est pas définie.

Les boucles Tant que :

```
while (/*condition*/) {
  // Code de la boucle
}

while (true) {
  console.log('Bonjour')
}
// Résultat
/*
Bonjour
Bonjour
```

```
Bonjour
...
Jusqu'à l'infini, car true ne changera jamais
*/
let i = 5;
let j = 2;
// Tant que j est supérieur à 0
while (j > 0) {
  if (i > j) {
    i -= j
  } else {
    j -= i
  }
  console.log('i : ${i}\nj : ${j}\n')
}
// Résultat
/*
i : 3
j : 2

i : 1
j : 2

i : 1
j : 1

i : 1
j : 0
*/
// j n'est plus supérieur à 0
```

Les boucles Pour

```

```

```
for (/*'variable de départ'*/; /*'condition de fonctionnement'*/; /*'expression de fin de boucle'*/) {
  // Code de la boucle
}
// Pour...
// i est la variable de départ
// Il faut que i soit inférieur à 5
// On incrémente i à la fin de la boucle
for (let i = 0; i < 5; i++) {
  console.log(i)
}
// Résultat :
/*
0
1
2
3
4
*/
```

Avec la boucle Pour, on peut par ailleurs parcourir une liste, avec 2 mots clés différents : `in`, qui va renvoyer à chaque itération de la boucle l'index auquel on se trouve, et `of` qui lui va renvoyer les valeurs.

```
let liste = ['Bonjour', 'Salut', 'Hey']
for (let el of liste) {
  console.log(el)
}
// Résultat
/*
Bonjour
Salut
Hey
*/
for (let i in liste) {
```

```
  console.log(i)
}
// Résultat
/*
0
1
2
*/
```

Je rappelle qu'il est possible de retrouver l'élement si on a l'index (`liste[i] = el`).

En appliquant la même logique qu'avec les `if`, nous pouvons utiliser une boucle Pour sans bloc :

```
for (let el of ['One', 'Two', 'Three']) console.log(el);
```

5. Les fonctions

▲

Définition

On dit d'une fonction qu'elle est définie, et qu'elle peut être appelée. La définition de la fonction correspond aux actions qui vont être executées quand elle va être appelée. Une définition seule ne va rien produire.

Il y a trois manière de définir une fonction :

```
/*
function nomDeLaFonction(arguments) {
  code executé à l'appel de la fonction
}
*/
function ditBonjour() {
```

```
  console.log('Bonjour')
}
ditBonjour() // => Bonjour
```

ou la version "stockage"

```
/*
const nomDeLaFonction = (arguments) => {
  code executé à l'appel de la fonction
}
*/
const ditBonjour = function() {
  console.log('Bonjour')
}
ditBonjour() // => Bonjour
```

ou la version "function fléchée"

```
/*
const nomDeLaFonction = (arguments) => {
  code executé à l'appel de la fonction
}
*/
const ditBonjour = () => {
  console.log('Bonjour')
}
ditBonjour() // => Bonjour
```

Ces deux denières méthode sont intéressantes, car elles impliquent et rappellent un élement fondamental de JS, qui est que tout un est un objet. `() => {}` et `function() {}` sont des fonctions, et dans les cas présentés au dessus, on stocke ces fonctions dans la constante `ditBonjour`, ce qui nous permettra de les

executer à l'avenir. Attention il y a cependant une différence entre une fonction fléchée (`() => {}`) et une fonction traditionnelle (`function name() {}` ou `function() {}`) : Outre la syntaxe plus courte, la fonction fléchée ne transmet pas this (présenté ultérierement).

Pour continuer dans la définition on peut aussi donner des arguments à une fonction :

```
function ditMoiBonjour(nom) {
  console.log('Bonjour ' + nom)
}
ditBonjour('Pierre') // => Bonjour Pierre
```

Jusqu'alors, nos fonctions envoyais quelque chose dans la console, une fonction peut aussi renvoyer une information, à stocker dans une variable :

```
function monNomAlenvers(nom) {
  let mon = ''
  for (let letter of nom) {
    mon = letter + mon
  }
  return mon
}
let result = monNomAlenvers('Pierre')
console.log(result) // => erreiP
```

Le mot clé `return` renvoie ce qui est derrière, et nous le stockons dans une variable que nous avons sobrement appelé result. Ce mot clé arrête d'ailleurs l'execution de la fonction :

```
function testing() {
  let i = 13
  return 'Hello ' + i
  i += 13
  console.log(i)
```

```
}
console.log(testing()) // => Hello 13
```

Tout ce qui était après `return` ne s'est pas executé.

Voici un exemple d'utilisation partique :

```
function inverse(a, b, c, d) {
    let det = a*d-b*c
    if (det === 0) return ['Invalide', []]

    let matrice = [
      [d*(1/det), -b*(1/det)],
      [-c*(1/det), a*(1/det)]
    ]

    return ['Valide', matrice]
  }

console.log(inverse(1, 2, 3, 4)) // => [ 'Valide', [ [ -2, 1 ], [ 1.5, -0.5 ] ] ]
console.log(testing(1, 4, 0.5, 2)) // => [ 'Invalide', [] ]
```

À des fins de précisions, il est tout à fait possible d'avoir une fonction qui fait un retour sans arguments. Il y a en fait quatre cas de figure :

	Avec retour	**Sans retour**
Avec argument	`let result = func(13)`	`func(13)`
Sans argument	`let result = func()`	`func()`

Héritage et `this`

Nous pouvons stocker nos fonctions, comme tout objet, dans des variables, mais aussi dans des liste, ou des dictionnaire, à côtés d'autres valeurs :

```
const list = [
    () => {
        console.log('one')
    },
    (two) => {
        console.log(two)
    },
    'hello'
]

list[0]()
list[1]('Salut')
list[1](list[2])
```

Ces fonctions sont considérées comme des fonctions héritées de l'objet liste, des méthodes de liste. Cet exemple permet de mieux intégrer des concepts comme ceux des méthodes de la `console` ou de l'objet `Math`. Les fonctions `log()` et `table()` sont des méthodes hérités de l'objet `console`, et les fonctions `floor()` et `random()` sont des méthodes hérités de l'objet `Math`. Pour aller encore plus loin, définisson ce qu'est un prototype, avec un exemple : Le prototype de `String`, d'une chaîne de caractère est un objet auquel on a attaché des méthodes, tel que `replace()`, ou encore `trim()`. Chaque chaîne de caractère obtient toutes ces mêmes méthodes.

```
console.log('Salut les copains'.replace('copains', 'amis')) // Salut les amis
```

En conséquence, nous pouvons y attacher de nouvelles méthodes (attention à ne pas utiliser un nom de fonction déjà existante) :

```
String.prototype.why = function() {
    // Obtenir une copie de ce qu'était la chaine de caractère
    let str = this.valueOf();
    // On y ajouter quelque chose
```

```
    str = str + ' ?';
    // Et on oublie pas de le renvoyer
    return str
}

console.log('salut') // salut
console.log('salut'.why()) // salut ?
```

Nous trouvons d'ailleurs un nouveau mot clé : `this`, qui fait référence à l'objet duquel on appel la fonction, l'objet qui hérite du prototype si la fonction y est attachée, ou simplement la liste ou le dictionnaire dans laquelle la fonction est stockée. Pour rappelle, ce mot clé n'est transmis qu'avec les fonctions traditionnelles (`function name() {}` ou `function() {}`), et pas dans une fonction fléchée (`() => {}`). Dans le cas ou une fonction est directement à la racine d'un fichier (ou imbriquée dans une autre fonction), this correspondra à l'objet global

"Qu'est-ce qu'un objet global en JavaScript ?

L'objet global en JavaScript est un objet toujours défini qui fournit des variables et des fonctions, et est disponible n'importe où. Dans un navigateur Web, l'objet global est l'objet window, alors qu'il est nommé global dans Node.js. L'objet global est accessible à l'aide de l'opérateur this dans la portée globale." (source)

```
Array.prototype.superPush = function(el) {
    if (!el) return this
    this.push(el)
    return this
}

Object.prototype.superSet = function(key, value) {
    if (!key || !value) return this
    this[key] = value
    return this
}

console.log([1].superPush(2)) // [ 1, 2 ]
console.log([1].superPush(null)) // [ 1 ]

console.log({salut:1}.superSet('hello', 2)) // { salut: 1, hello: 2 }
```

```
console.log({salut:1}.superSet(null, 2)) // { salut: 1 }
console.log({salut:1}.superSet('hello', null)) // { salut: 1 }
```

Dans cette exemple, toute liste présente après cette définition pourra profiter de la méthode `superPush()`, et tout dictionnaire après cette définition pourra profiter de la méthode `superSet()`.

```
const dict = {
    one: function() {
        console.log(this.salut)
        this.salut = 'non'
    },
    salut:'oui'
}

dict.one() // => oui
dict.one() // => non
```

Ici `this` correspond au dictionnaire `dict`, donc `this.salut` correspond à `dict.salut`. Comme la fonction est attachée à l'objet et non pas au protoype `Object.prototype` (ou `Array.protype` dans le cas d'une liste), cette méthode ne sera hérité que de cette objet.

Dans le cas du stockage d'une méthode dans une fonction, ```this`` fait référence au palier actuel, donc il renverra toujours le dictionnaire, ou la liste, dans lequel il est stocké :

```
const dict = {
    one: function() {
        console.log(this.salut)
    },
    salut:'oui',
    inner:{
        func: function() {
            console.log(this.salut)
        },
```

```
        salut:'non'
    }
}

dict.one() // oui
dict.inner.func() // non
```