Cédric Fontaine

JavaScript

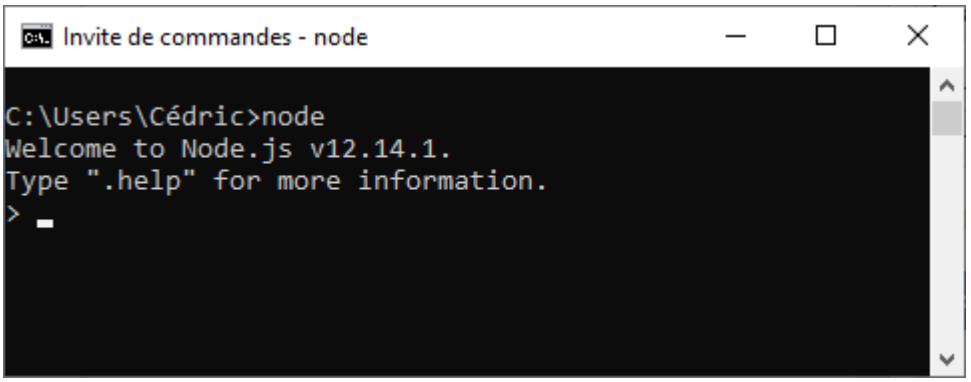
Définition

JavaScript est initialement un langage de programmation web, interprété en temps réel, utilisé pour rendre les pages dynamiques. Il permet de manipuler le **DOM** et d'ajouter, supprimer, éditer son contenu. Depuis quelques années maintenant, il est possible d'utiliser le JavaScript dans de nombreux autres domaines, du rendu serveur au développement d'applications mobile et bureau, grâce à **NodeJS**.

DOM (Document Object Model) est l'interface qui rend le **HTML** compréhensible par JS. La structure de la page est un arbre composé de nœuds agencés hiérarchiquement (e.g: le nœud **html** est la racine de la page, et le parent de **body**, lui-même parent d'autres nœuds, etc.).

Utilisation

Il est possible de coder en JavaScript depuis un éditeur de texte, voire directement depuis la console de développement intégré à n'importe quel navigateur moderne. Les raccourcis les plus courants sont **F12** et **CTRL + SHIFT + K**.
Si NodeJS est installé sur la machine (par défaut sur les systèmes Linux récents), il suffit de taper la commande **node** dans un terminal pour ouvrir un environnement de développement.
Sinon, il faut l'installer puis l'ajouter dans la variable d'environnement **PATH**.
La commande **node fichier.js** exécute un programme JavaScript dans cet environnement.



Hello World !

L'outil **Console** est un "Objet Intégré" (*Built-in feature*), qui permet d'afficher toutes sortes d'informations, principalement à des fins de débogage. Par défaut, on utilise la fonction "log".

```
console.log("Hello world!");
```

Variables

JavaScript est un langage de haut niveau avec un typage faible. À contrario des langages de bas niveau, il n'est pas nécessaire de préciser le type des données stockées. On déclare une variable avec le mot clé ***var***. Du fait de la présence de la *Garbage Collection* (en bon Français, le ramasse-miettes), la gestion de la mémoire est automatisée.

Pros	Cons
Rend la programmation plus accessible.	Il est nécessaire de toujours savoir à quoi correspondent les données.

```
var maVariable = 42; // notation camelCase
console.log(maVariable);
// => 42
maVariable = "Foobar";
console.log(maVariable);
// => "Foobar"
```

Types

Les structures de données natives en JavaScript sont les suivantes.

Type	Spécificité	Exemple
Boolean()	{ true | false }	`var estMajeur = true;`

null	Indique l'absence de valeur d'une variable à moment donné. (Généralement utilisé pour les propriétés d'objets)	`var record = null;`
		`console.log(record);`
		`// => null`

undefined	Représente une donnée non initialisée (qui n'a pas de valeur).
	
	/!\ Le mot *undefined* en soit est quand même une valeur.
		`var prenom;`
		`console.log(prenom);`
		`// => undefined`

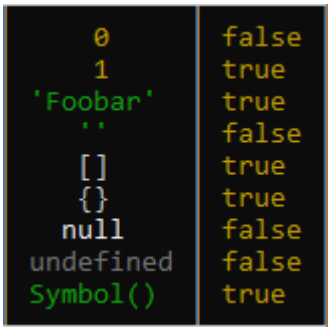
String()	Représentation du texte. Regroupe à la fois un caractère simple, et les chaînes de caractères.	`var prenom = "Foobar";`
		`console.log(prenom);`
		`// => "Foobar"`

Number()	[$-(2^{53} - 1)$; $(2^{53} - 1)$]. Regroupe à la fois les entiers et les nombres à virgule flottante.	`var age = 21;`
		`console.log(age);`
		`// => 21`

Object	Structure de données possédant des propriétés (*fields*), et des fonctions (*methods*).	`var joueur = {`
		`  "nom": "Foobar",`
		`  "record": null`
		`}`
		
		`console.log(joueur["nom"]);`
		`// => "Foobar"`
		`console.log(joueur.record);`
		`// => null`

Symbol	(Récent)
	Initialement créé pour donner un contexte privé aux propriétés d'un objet, un symbole représente une donnée unique.
	La valeur d'un symbole est générée dans un environnement global dédié et est assurément unique.
		`var sA = Symbol();`
		`var sB = Symbol();`
		
		`console.log(sA === sB);`
		`// => false`

Backus-Naur Form



Toute valeur a une représentation booléenne, et donc toute expression renvoie une valeur booléenne. Une expression est une phrase de la forme:

{ valeur } { verbe } { valeur }

Où le verbe est l'opérateur de comparaison, et une valeur peut également être une expression:

```
age == 21

age: valeur
==: verbe
21: valeur
```

```
age == 7 * 3

age: valeur
== : verbe

7 * 3: expression avec

7: valeur
*: verbe
3: valeur
```

Une phrase est une suite d'opérations exécutées jusqu'à obtenir une valeur dite «finie», ou «immuable».

```
age >= 21 && age <= 60

age: valeur
>=: verbe
21: valeur

&&: verbe (connecteur logique)

age: valeur
<=: verbe
60: valeur
```

Comparaison

Il existe plusieurs “verbes” de comparaison et connecteurs logiques:

{ < | > }	Plus petit que Plus grand que
{ <= | >= }	Plus petit ou égal à Plus grand ou égal à
==	Comparaison souple
===	Comparaison stricte
&&	AND
||	OR
!	NOT

Les structures de contrôles sont définies par les mots-clés ***if… else if… else***.

```
if (condition) {
 // ...
} else if (condition) {
 // ...
} else {
 // ...
}
```

Ou bien un **switch**, généralement utilisé pour vérifier des valeurs énumérables.

```
var direction = getDirection();

switch(direction) {
  case 'up':
    // ...
    break;
    // Un switch s'exécute en cascade. La présence du break permet de
sortir du switch sans "tomber" dans le cas suivant.
  case 'down':
    // ...
    break;
  default: // Aucune des conditions ci-dessus n’est satisfaite
    // ...
}
```

Opérations ternaires

Une opération ternaire est une manière plus courte d'écrire une structure de contrôle ***if … else***.
Elle se lit comme une question, avec pour réponse 2 expressions de retour selon le booléen renvoyé par la condition, séparés par ":".

Question ? true : false

```
var age = 17;
var message = (age >= 18 ? "majeur" : "mineur");
console.log("Tu es " + message);
// => "Tu es mineur";

age = 21;
var message = (age >= 18 ? "majeur" : "mineur");
console.log("Tu es " + message);
// => "Tu es majeur";
```

Boucles

Il est possible de répéter du code avec différentes boucles:

```
while (condition) {
  // ...
}

do {
  // ...
} while (condition);

for (début; fin; itération) {
 // ...
}

for (propriété in objet) { // itère sur toutes les propriétés d'un objet
 // ...
}

for (itérateur of iterable) { // itère sur toutes les valeurs d'un objet
de type tableau (array)
 // ...
}
```

Fonctions

Les fonctions sont des morceaux de codes réutilisables permettant de traiter des données, et moduler un programme. Elles sont déclarées avec le mot clé ***function*** et délimitées par des accolades, entre lesquelles est déclaré le code a exécuter.

```
function hello() {
  console.log("Hello world!");
}

hello();

// => "Hello world!"
```

Elles peuvent recevoir des paramètres de manière souple (c'est à dire qu'on peut y passer un nombre différent des paramètres déclarés dans la signature), en utiliser les valeurs, et retourner un résultat.

/!\ Un affichage n'est pas un retour de valeur /!

```
function hello(prenom) {
  console.log("Hello " + prenom); // Affiche "Hello <insérer prenom>" sans la retourner
  return "Hello " + prenom; // renvoie effectivement la chaine de caractères
}

console.log(hello("Foobar"));
// => "Hello Foobar" | premier affichage dû à la ligne 3
// => "Hello Foobar" | affichage de la valeur retournée par la fonction
```

S'il est possible de nommer une fonction à sa déclaration, on peut également attribuer une fonction à une variable (*function expression*). Une fonction déclarée est chargée avant l'exécution du code, tandis qu'une *function expression* est chargée uniquement lorsque l'interpréteur se trouve sur sa ligne.

```
var hello = function () { // fonction anonyme
  return "Hello world!";
}

console.log(hello);
// => ∟ function()

console.log(hello());
// => "Hello world!"
```

Par ailleurs, on peut également créer une fonction qui s'exécute elle même (pour protéger l'environnement des données utilisées, par exemple).

```
(function maFonction([param1, [param2, [...) {
  // ...
})([arg1, [arg2, [...);
```

Arguments

Il est possible d'accepter dans une fonction un nombre indéfini de paramètres, et d'y accéder depuis un objet natif semblable à un array appelé ***arguments***.

```
(function () {
  for (var i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
})("Hello", "World", "Foo", "bar");

// => "Hello"
// => "World"
// => "Foo"
// => "bar"
```

De plus, les fonctions peuvent accepter des paramètres "par défaut", avec une valeur de "retrait" si aucun argument n'est fourni.

```
function hello (prenom = "inconnu") {
  console.log("Salutations " + prenom + ".");
};

hello();
// => "Salutations inconnu."

hello("Foobar”);
// => "Salutations Foobar."
```

Callbacks

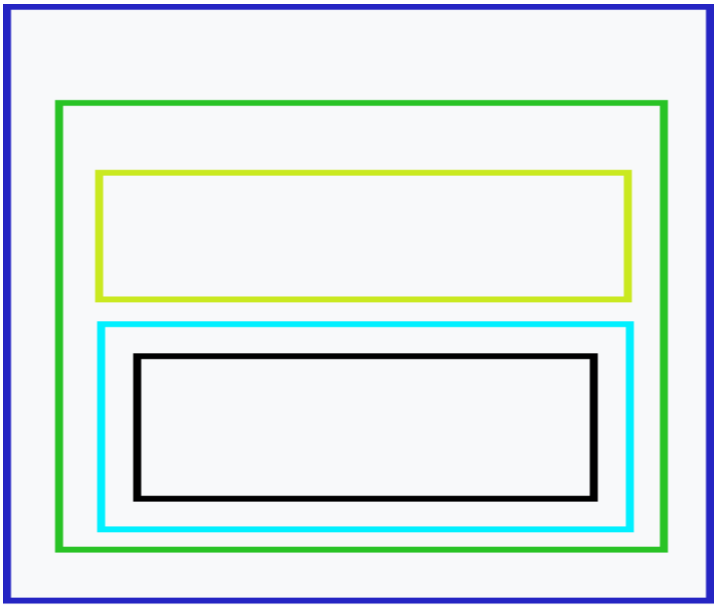
Un callback est une fonction dite «de rappel», passée en paramètre dans, et utilisée par une autre fonction.

```
function hello(nom) {
  return "Hello " + nom;
}

function maFonction(callback) {
  console.log(callback("Foobar"));
  // callback est exécutée, son résultat est affiché par maFonction
}

maFonction(hello);
// => "Hello Foobar"
```

Contextes (scope)



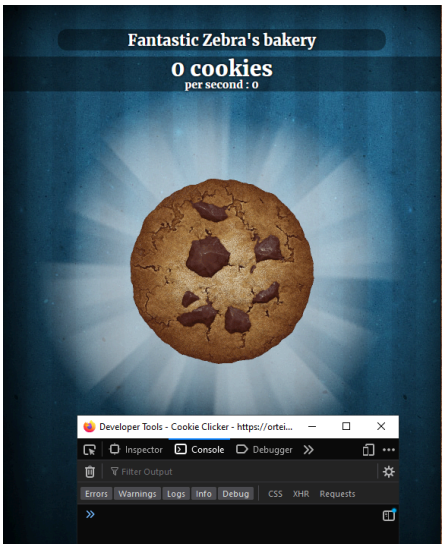
Un contexte détermine la visibilité des variables et fonctions. L'environnement le plus large est le **global**. Une variable déclarée dans l'environnement global est accessible partout dans le code. Grâce aux fonctions, il est possible de limiter l'environnement à un contexte local, inaccessible depuis l'extérieur, et qui a accès aux contextes dans lesquels il est défini.

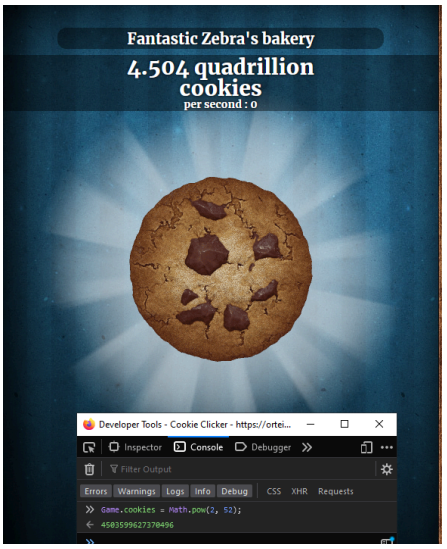
Ici, le contexte **global** n'a pas accès au premier contexte **local**, bien que le contraire soit possible, et ainsi de suite.

Cela peut par exemple servir à protéger l'accès aux données d'une page web:

```
// script_de_mon_site.js
(function() {
  // ...
})();
```

Encapsuler la totalité du script permet de créer un contexte local auquel un visiteur, qui se trouve dans le contexte global, n'aura pas accès depuis la console de développement.
Exemple d'un site non "sécurisé".
Si le moteur du jeu avait été encapsulé, l'usage de ses fonctions serait impossible.





Générateurs

Un générateur est un objet itérable et itérateur. Il s'agit d'une fonction dont les instructions peuvent avoir des **breakpoints** définis par le mot clé **yield**. Les instructions sont exécutées jusqu'à rencontrer un breakpoint qui met la fonction en pause. On peut passer à la suite en appelant la méthode **.next()** sur le générateur. Chaque breakpoint retourne une valeur à laquelle on accède avec l'attribut **.value**.

```
function* hellos() {
  yield "Hello";
  console.log("Foobar");
  yield "Hi";
  yield "Hello world";
}

var gen = hellos();

console.log(gen.next().value);
// => "Hello"

console.log(gen.next().value);
// => "Foobar" | Affichage de Foobar
// => "Hi" | Affichage de la valeur retournée par le breakpoint

console.log(gen.next().value);
// => "Hello world"
```

Arrays

Un array est une structure de données itérable représentant un tableau à dimensions variables. C'est un prototype d'objet natif pouvant contenir tous types de données, déclaré non pas par des accolades, mais des crochets. Cet objet possède nombre de méthodes permettant de le manipuler. On accède à ses éléments grâce à leur indice (position) dans le tableau (les indices commencent à 0 et vont jusqu'à N-1, N étant la taille du tableau).

```
var table = [7, 3, 45, "Hi", true, false, {}, function() {return 12}];
console.log(table.length);
// => 8

console.log(table[0]);
// => 7

console.log(table[table.length]);
// => undefined | Il n'y a pas d'élément en 8ème position.

console.log(table[table.length - 1]());
// => 12
```

Pour parcourir un tableau, il suffit d'accéder à chaque élément avec une boucle.

```
for (var i = 0; i < table.length; i++) {
  console.log(table[i]);
}
// ou
for (var i of table) {
  console.log(i);
}
```

ECMAScript

L'ECMAScript (ou ES) est le standard de version de JavaScript.

Chaque édition apporte son lot de *Built-in features*, facilitant grandement la production, l'une des plus importantes étant l'ES6.

.map()

.map() retourne un tableau contenant les valeurs modifiées du tableau sur lequel on l'applique.

```
// On envoie la valeur et l'indice de chaque élément du tableau, ainsi
que le tableau lui-même dans la fonction passée dans la méthode map.
// Le résultat de chaque itération retourné par cette fonction est
inséré dans le tableau vide, qui est retourné à la fin de la méthode.

/**
 * Fonction qui double la valeur de chaque élément d'un tableau
 */
function doubler(element, index, array) {
  return element * 2;
}

/**
 * Cette méthode est native, il n'est pas nécessaire de la déclarer,
mais si on voulait la créer "from scratch", ça ressemblerait à ça :
 */

Array.prototype.map = function(callback) {
  var array = [];
  // this représente le tableau sur lequel on appelle la méthode .map()
  for (var i = 0; i < this.length; i++) {
    array.push(callback(this[i], i, this));
  }
  return array;
}

// On appelle la méthode map de l'objet
var array = [1, 2, 3, 4, 5].map(doubler);
console.log(array);
// => [2, 4, 6, 8, 10]
```

.filter()

.filter() est une méthode déclarative permettant de filtrer un tableau, en créant un nouveau tableau ne contenant que les valeurs respectant une certaine condition.

```
/**
 * Fonction qui renvoie true si le paramètre element est pair
 */
function estPair(element, index, array) {
  return !(element % 2);
}

Array.prototype.filter = function(callback) {
  var array = [];
  // this représente le tableau sur lequel on appelle la méthode map.
  for (var i = 0; i < this.length; i++) {
    if (callback(this[i], i, this)) {
      array.push(this[i]);
    }
  }
  return array;
}

// On appelle la méthode filter de l'objet
var array = [1, 2, 3, 4, 5].filter(estPair);
console.log(array);
// => [2, 4, 6, 8, 10]
```

.forEach()

.forEach() exécute une fonction à chaque itération sur les éléments d'un tableau. Elle ne retourne pas de résultat, et n'est donc généralement pas utilisée pour modifier le tableau.

```
function afficherValeur(element, index, array) {
  console.log(element);
  => Affiche la valeur du paramètre élément.
}

Array.prototype.forEach = function(callback) {
  for (var i = 0; i < this.length; i++) {
    callback(this[i], i, this);
  }
}

[1, 2, 3, 4, 5, 6].forEach(afficherValeur); // Remarquez qu'on n'affecte
aucune valeur.
/*
=> 1
=> 2
=> 3
=> 4
=> 5
=> 6
*/
```

.reduce()

.reduce() “accumule” des valeurs obtenues après une série d’instructions, et réduit le tout en une valeur finie.

```
// Cette fonction retourne une somme
function somme(accumulateur, element, index, array) {
  return accumulateur + element;
}

// reduce reçoit la fonction somme en callback, et un paramètre (0 par
défaut)
Array.prototype.reduce = function(callback, k = 0) {
  // La variable résultat accumulera la somme des éléments à chaque
itération
  var resultat = k;
  for (var i = 0; i < this.length; i++) {
    resultat = callback(resultat, this[i], i, this);
  }
  return resultat;
}

// On passe la fonction somme en callback de la méthode reduce
var sum = [1, 2, 3, 4, 5].reduce(somme);
console.log(sum);
// => 15

var reduce = [1, 2, 3, 4, 5].reduce(function(accumulateur, element,
index, array) {
  return accumulateur * element;
}, 10);

console.log(reduce);
// => 1200 // 10 * 1 * 2 * 3 * 4 * 5
```

Template literals

Ou *littéraux de gabarits* sont des chaînes de caractères permettant d'intégrer des expressions en tant que "jetons" dont les valeurs prennent retournées la place. Ils sont limités par des ***backticks*** (**ALT GR + 7** → `), et les jetons par le format **${**expression**}**.

```
var prenom = "Foobar";
console.log(`Bonjour ${prenom} !`);
// => "Bonjour Foobar !";
```

De plus, ils offrent la possibilité de formater du texte sur plusieurs lignes sans avoir recours à des concaténations ou "retours chariots" (\n).

```
console.log(`Hello world
My name is Foo.`);

// => "Hello world
// My name is Foo."
```

Arrow functions

Les fonctions fléchées sont un autre raccourci proposé par ES6. Il s'agit d'une *function expression* pouvant tenir sur une seule ligne (retournant une unique instruction), ou contenir un bloc de code standard.

```
var hello = ([param1, [param2, [...) => {
  return "Hello world!";
});
// équivaut à

var hello = function([param1, [param2, [...) {
  return "Hello world!";
}
// équivaut à

var hello = ([param1, [param2, [...) => "Hello world!";
// En l'absence d'accolades, la fonction ne peut contenir qu'une seule
instruction, et omet le keyword "return".

console.log(hello([arg1, [arg2, [...));
// => "Hello world!"
```

Exemple pratique avec les méthodes précédentes

```
var tableau = [1, 2, "a", "b", 3, 4,"c", "d",5, 6,"e", "f",7,
8,"g","h"];

tableau = tableau
    .filter((element) => ! isNaN(element)) // retourne un array ne
contenant que des nombres
    .map((element) => element * 2); // double les valeur du tableau
filtré et l'affecte à la variable tableau

tableau.forEach((element) => console.log(element));
/*
=> 2
=> 4
=> 6
=> 8
=> 10
=> 12
=> 14
=> 16
*/

console.log(tableau.reduce((accumulateur, element) => accumulateur +
element, 0));
// => 72
```

Décomposition

La décomposition (*destructuring*) permet d'extraire les données et propriétés d'une structure de données itérable et de les étendre sur plusieurs variables.

```
var array = ["Hello", "world", "foo", "bar"];

// syntaxe littérale
var [a, b, c, d] = array;
// S'il y a plus de variables que d'éléments dans le tableau, les
variables supplémentaires recevront la valeur undefined
// S'il y a plus d'éléments dans le tableau, seules les premiers
éléments seront affectés aux variables

console.log(a, b, c, d);
// => "Hello world foo bar"

// syntaxe de répartition
function hello(... params) {
  console.log(params);
}
hello(... array);
hello(a, b, c, d);
```

Exemple concret du spread operator

```
// Tous les arguments passés dans la fonctions sont traités comme un
unique array, sans qu'il ne soit nécessaire de préciser un nombre exacte
de paramètres.
function addition(...params) {
  return params.reduce((a, e) => a + e, 0);
}

console.log(addition(3, 2, 0, -7, 14, 42, 66, 34));
// => 154
```