

Variables, types et opérateurs

Table des matières

- [Avant propos](#)
- [Comment suivre ce chapitre](#)
- [Java et Javascript](#)
- [Etape 1 - Les variables et les constantes](#)
 - [const](#)
 - [Comportement](#)
 - [Convention de nommage des constantes](#)
 - [Quelques mauvais exemples](#)
 - [let](#)
 - [Comportement](#)
 - [Conventions de nommage](#)
 - [Quelques mauvais exemples](#)
 - [var](#)
 - [Comme "let"](#)
 - [Mais en différent](#)
 - [Quand utiliser const, let et var](#)
- [Etape 2 - Les types et leurs opérateurs](#)
 - [Les types primitifs](#)
 - [Boolean](#)
 - [Null](#)
 - [Undefined](#)
 - [Number](#)
 - [Chaines de caractères](#)
 - [Les types complexes](#)
 - [Les tableaux](#)
 - [Les objets "libres"](#)
 - [Les objets structurés](#)

Avant propos

Dans ce tout premier chapitre, nous allons découvrir le fonctionnement de JavaScript d'une manière peut-être déstabilisante. Javascript est un langage initialement créé pour être exécuté au sein d'un navigateur et la logique voudrait que nous commençons immédiatement à manipuler ces concepts là. Pourtant avant toute chose, je vous propose d'un peu jouer avec les variables, les fonctions, la syntaxe etc. Pourquoi ce parti pris ? Parce qu'il n'est pas souhaitable de prendre de mauvaises habitudes et qu'une compréhension solide de ce langage très permissif vous permettra de travailler bien plus sereinement.

Le but de ce cours est de comprendre que JavaScript est un vrai langage de programmation, avec ses règles très précises, avec ses concepts très clairement décrits. Cela n'était pas forcément vrai il y a quelques années et de mauvais réflexes quant à son apprentissage doivent disparaître.

Ces changements - *pour le mieux nous en conviendrons* - sont dus à ECMAScript, qui est une spécification qui a pour vocation à décrire comment JavaScript devrait fonctionner. En 2015, la 6ème édition de ce format est apparue, amenant son lot de mise au propre et de grandes améliorations. Depuis, nous en sommes à la 9ème édition (1 par an !), ce qui a pour conséquences une amélioration (et une complexification ?) considérable du langage.



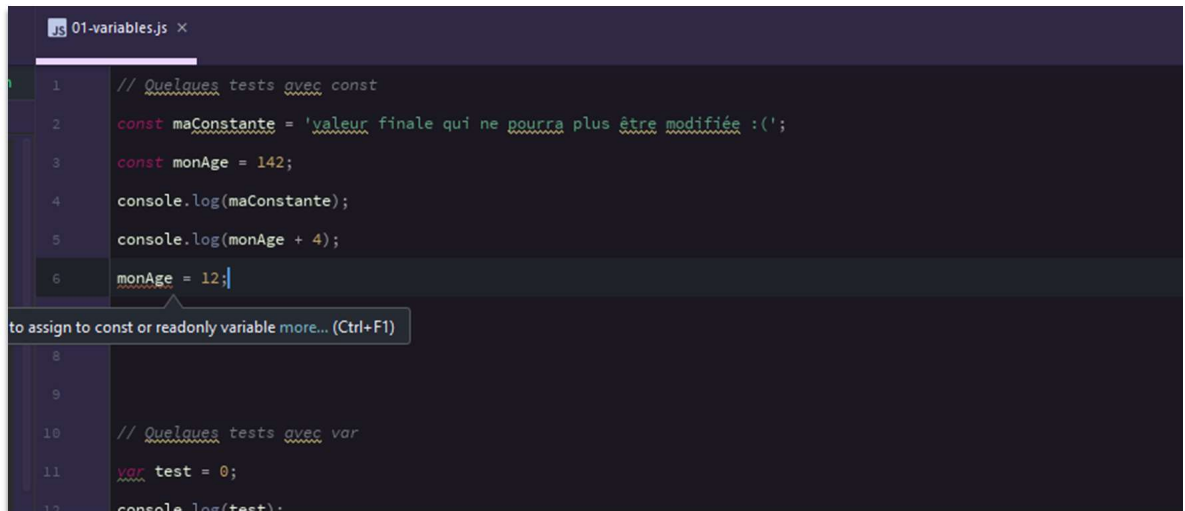
ECMAScript et ses versions

Nous ne nous perdrons pas dans trop de digressions au sujet d'ECMAScript, du support des versions récentes de JavaScript dans les différents navigateurs, etc. mais cela peut être très intéressant, aussi je vous invite sérieusement à jeter un oeil à tout cela.

Comment suivre ce chapitre

Vous pouvez simplement le lire mais ce n'est vraiment pas conseillé. Vous devriez toujours éprouver ce que je vous propose. Testez ces morceaux de code, modifiez les, testez les cas improbables !

Pour mettre tout cela en pratique, nous avons simplement besoin d'un navigateur, et pourquoi pas d'un éditeur de texte pour que vous puissiez copier/coller votre code voire même l'exécuter si l'edit IDE rockse sa maman :



```
01-variables.js x
1 // Quelques tests avec const
2 const maConstante = 'valeur finale qui ne pourra plus être modifiée :(';
3 const monAge = 142;
4 console.log(maConstante);
5 console.log(monAge + 4);
6 monAge = 12;
7
8
9
10 // Quelques tests avec var
11 var test = 0;
12 console.log(test);
```

to assign to const or readonly variable more... (Ctrl+F1)

Commencez donc par ouvrir votre navigateur préféré et ouvrez les outils de développement qu'il met à disposition

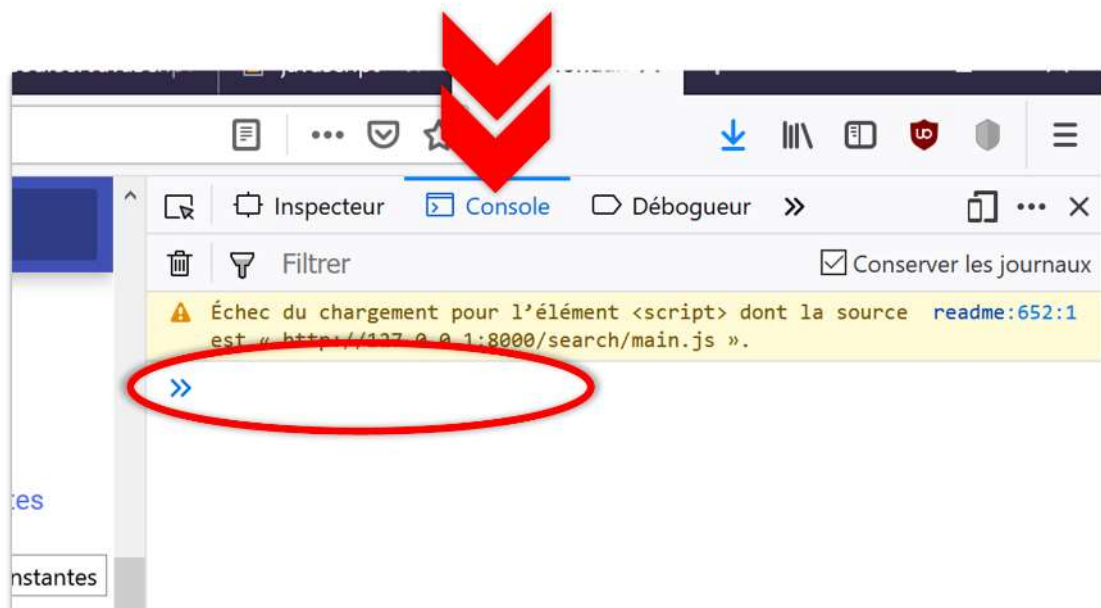
☰ Ouvrir les outils de développement

Dans la plus part des navigateurs, vous pouvez ouvrir vos outils de développement à l'aide d'un des deux raccourcis suivants :

- <maj> + <ctrl> + <i>
- <F12>

Selon votre navigateur et votre paramétrage, cette nouvelle fenêtre pourra prendre différentes apparences, mais cela devrait peu ou prou ressembler à l'image ci-dessous.

Cliquez d'abord sur l'onglet "console", et vous disposez alors d'un prompt, c'est à dire une box dans laquelle vous pouvez saisir toutes vos commandes, particulièrement dans le cas de ce premier cours.



Java et Javascript

⚠ Danger

Ce ne SONT PAS les mêmes langages.

Chaque fois que vous utilisez un mot à la place de l'autre, le réchauffement climatique tue un gentil ours polaire. Oui, cette information est suffisamment importante pour bénéficier de son propre chapitre.

Etape 1 - Les variables et les constantes

Commençons donc immédiatement par une spécificité du langage qui ne nous offre pas moins de 3 mots clés rien que pour définir des boîtes dans lesquelles nous pouvons stocker des informations ! Et nous allons les découvrir dans l'ordre qui me semble le plus pertinent

const

Lorsque l'on veut stocker une information dans une variable, la première question à se poser est "Devrais-je modifier cette information plus tard ?". Si la réponse est "Non", vous pouvez utiliser une constante et cela se passe de la façon suivante, avec le mot clé `const` :

```
const maConstante = 'valeur finale qui ne pourra plus être modifiée :(';  
const monAge = 142;
```

Comportement

En faisant cela, nous avons créé deux constantes, une qui est une chaîne de caractère et une qui est un nombre. Nous reviendrons sur cela dans peu de temps. Ce qu'il faut retenir, c'est que vous pouvez manipuler ces deux objets pour en lire le contenu mais pas pour les modifier :

```
console.log(maConstante);  
console.log(monAge + 4);  
monAge = 12;
```

aura pour résultat :

```
valeur finale qui ne pourra plus être modifiée :(  
146  
C:\Users\...\01-variables.js:6  
monAge = 12;  
      ^  
TypeError: Assignment to constant variable.
```

Vous constatez ici que parmi les 3 commandes que nous avons exécuté, les deux premières se sont bien déroulées et affichent bien le résultat que l'on est en droit d'attendre. Par contre, lorsque l'on souhaite modifier notre âge, une erreur survient. Ceci est logique, aussi bien en termes informatiques qu'en termes biologiques d'ailleurs. Cela étant, il serait fort gênant de ne pouvoir utiliser que des constantes.

Convention de nommage des constantes

Il n'existe pas forcément de véritables règles quant aux conventions de nommage en JavaScript mais il existe une sorte de consensus que nous nous efforcerons d'adopter :

les constantes non calculées devraient être nommées en majuscules

```
const VERITE = '42';
const PI = 3.14159265358979323846264
const SITE_URL = 'http://localhost'
```

Les constantes résultant d'un calcul devraient l'être en minuscule

```
const begin = new Date();
const surface = 2 * PI * 12;
```

Quelques mauvais exemples

```
const Politesse = 'Hello';           // nom en titlecase => passe en MAJUSCULES
const CONSTATENTEENPLUSIEURSMOTS = 'bonjour'; // mots à séparer par des underscores
const pi = 3.14;                      // propriété connue avant l'exécution
const BEGIN = new Date()              // propriété non connu avant l'exécution
```

let

Lorsque l'on souhaite déclarer des variable en JavaScript, `let` est certainement la manière que vous devriez systématiquement considérer. Vous allez constater que leur usage est similaire à celui des constantes :

```
let maVariable = "bobby";
let nb1 = 10;
let nb2 = 12;
console.log(maVariable);
console.log(nb1);
```

Comportement

En exécutant ce code, vous pouvez constater que le comportement est en tout point similaire à celui des constantes. Maintenant, nous allons voir ce qui se passe en cas de **réaffectation** :

```
maVariable = "override dat!";
nb1 = nb2 + 42;
console.log(maVariable);
console.log(nb1);
```

Vous constatez alors que cette fois vous pouvez modifier les valeurs de vos variables et c'est tant mieux, car nous venons de voir les deux concepts que vous attendiez de trouver. Alors pourquoi en avoir un troisième ?

Conventions de nommage

Les noms de variables en JavaScript devraient systématiquement être écrits en `camel case` (haha ! Cherchez sur internet). Tous les noms de variables devraient commencer par une lettre, écrite en minuscule :

```
let test = "Bonjour";
let maVariableEnCamelCase = "Bonjour";
```

Quelques mauvais exemples

```
let VARIABLE = 'ma valeur';          // devrait être écrit en minuscules
let ma_variable = 10;                 // devrait être en camelcase
```

```
let _date = 12;           // devrait commencer par une lettre
let 2ndDate = 12;         // même chose
```

var

Historiquement, Javascript ne proposait que `var`. En ces temps funestes, il n'était pas question de parler de constantes et la portée de variables était complexe à gérer, mais nous reparlerons de tout cela plus tard.

Comme "let"

Au premier abord, `var` est similaire en tout point à `let` : on parle bien de variables, on les nomme de la même manière, on peut les modifier de la même manière. Pour autant nous noterons quelques différences et d'ailleurs, en dehors des restrictions dues à ces différences que nous allons exposer, vous ne devriez jamais utiliser `var`.

Mais en différent

 Pour aller plus loin

Ce que nous détaillons ici peut vous sembler bien trop complexe ou abscons. Ce n'est pas un souci, vous pouvez vous en passer et simplement considérer que le jour où vous aurez besoin de `var`, soit la directive n'existera plus, soit vous serez en l'état de comprendre, soit vous devrez reconsidérer sérieusement la structuration de votre code.

- lorsque l'on déclare une variable avec `var`, elle est accessible partout dans notre page comme attribut de l'objet `window` : `window.myVar`, ce qui n'est pas le cas avec `let`
- lorsque vous utilisez `let` dans un bloc, la portée de la variable se limite à ce block. Lorsque vous la déclarez avec `var`, elle reste disponible pour la suite de la fonction...
- on peut redéclarer une variable déclarée avec `var`, pas avec `let` :

```
var redefinable = 'première définition';
var redefinable = 'nouvelle définition !';

let noRedefinable = 'première définition';
let noRedefinable = 'nouvelle définition !';
var noRedefinable = 'nouvelle définition !';
```

Dans ce cas par exemple, la redéfinition de `redefinable` ne pose aucun souci. En revanche, la redéfinition de `noRedefinable` entraîne une erreur, que ce soit via l'utilisation de `var` ou `let` : `let` permet de 'bloquer' la réutilisation d'un nom de variable.

Quand utiliser const, let et var

Le moyen le plus simple est le suivant :

- essayez d'utiliser `const`
- si ça ne fonctionne pas, utilisez `let`
- si ça ne fonctionne pas, repensez votre code (ou utilisez `var` ...)

 Tester dans le navigateur et dans votre IDE

Nous allons voir comment tout cela s'utilise dans notre navigateur via l'utilisation des outils de développement du navigateur avec en préambule une présentation rapide de tous ces outils.

Dans un second temps, nous allons voir que tout cela peut aussi se faire dans certains IDE et que ce type de fonctionnalité est un véritable confort de développement et contribue grandement à établir des expérimentations et permet d'aboutir à des programmes de meilleures qualités !

Etape 2 - Les types et leurs opérateurs

Nous distinguons deux catégories de structure de données : les types primitifs qui sont la base qui va nous permettre de travailler et les types complexes, qui peuvent être vus comme une combinaison structurée de données primitives ou complexes. Vous comprenez à la lecture de cette phrase que l'on n'est plus en train de rire même si tout ceci est plutôt simple. C'est parti

Les types primitifs

Ils sont au nombre de 6 et nous nous contenterons d'aborder les 5 premiers dans un premier temps car le dernier exige la connaissance de concepts que nous n'avons pas encore manipulés.

Ces types sont :

Boolean

Une valeur booléenne permet d'exprimer le fait qu'une "chose" est vraie ou fausse. Cela s'utilise de la manière suivante :

```
const isRainyDay = false;
let isNewUser = true;
isNewUser = false;
```

On peut effectuer des opérations simples sur les booléens, via l'utilisation d'opérateurs :

- pour tester l'égalité `===`
- pour tester la "non égalité" `!==`
- pour inverser la valeur de notre variable `!`

```
const isRainyDay = false;
let isNewUser = true;
console.log(isNewUser);
// true
console.log(!isRainyDay);
// true
console.log(isRainyDay === isNewUser);
// false
console.log(isRainyDay !== isNewUser);
// true
isNewUser = !isNewUser
console.log(isNewUser);
// false
```

Nous verrons plus tard à quoi cela peut servir. Si vous avez des connaissances en développement, vous savez déjà tout cela, en tous cas, c'est peut-être ce que vous pensez. JavaScript est plein de surprises <3...

Null

Il est possible qu'une variable existe mais que l'on souhaite à un moment qu'elle n'ait pas ou plus de valeur. Par exemple, si on a un jeu de bataille navale, on souhaite stocker dans une variable le joueur actif, c'est à dire celui qui est censé avoir le droit de jouer à ce moment là. Mais si à un moment donné, aucun joueur n'a la main, par exemple avant le début de la partie, on peut souhaiter que cette variable soit nulle ou n'existe pas, et on fait cela à l'aide du type null :

```
var currentPlayer = "Bob";
currentPlayer = 'Alice';
currentPlayer = null;
console.log(currentPlayer);
```

Il n'y a pas d'opérateurs particulier sur ce type de données.

Undefined

Nous parlons de types de données et voilà le deuxième type dont on parle qui indique que nous n'avons pas de données... Et pourquoi u

deuxième type pour cela ? Parce que cette fois, le but est de tester l'existence d'une variable, pas de savoir si elle est nulle. Revenons à notre exemple de bataille navale. Lorsque l'on démarre une partie, un joueur doit renseigner ses informations (comme son identifiant par exemple). Dans ce cas là, on peut vouloir tester si l'utilisateur a renseigné son identifiant en le laissant vide (peut être nul dans ce cas) ou s'il n'a tout simplement pas encore défini son identifiant (indéfini dans ce cas). En usage normal, une variable est `undefined` si elle existe mais qu'on ne lui a pas encore donné de valeur :

```
var who;
console.log(who);
// undefined
```

Il n'y a pas d'opérateurs particulier sur ce type de données.

Number

Revenons à des types que l'on manipule souvent : les nombres. Il n'y a pas grand chose à dire sur le format lui même, jettons un peu un oeil à leur fonctionnement :

```
let n1 = 0;
let n2 = 12.5;
let n3 = .2457;
let n4 = -5.2;
console.log(n1);
// 0
console.log(n2);
// 12.5
console.log(n3);
// 0.2457
console.log(n4);
// -5.2
```

Vous remarquez que l'on peut ne pas préciser la partie entière d'un nombre, sa valeur est alors comprise comme 0. On peut créer des nombres entiers, des décimaux, en positif ou en négatif : simple mais efficace.

Chaines de caractères

Le dernier type primitif que nous allons aborder est la chaine de caractères. Aussi simple qu'il puisse sembler être, nous allons tout de même voir qu'il existe des subtilités qu'il vaut mieux connaître.

Pour représenter une chaine de caractères, vous pourrez utiliser indifféremment les `simple quotes` (apostrophes) et les `double quotes` (guillemets). TypeScript recommande d'utiliser des `simple quotes` en première intention et, même si vous ne savez pas encore ce que c'est, je vous recommande de prendre cette habitude également, mais ce n'est qu'un avis personnel. On utilise généralement l'un ou l'autre pour ne pas pouvoir inclure l'autre dans la chaine de caractère sans avoir à l'échapper. Sauvons vos neurones en vous proposant un exemple plutôt qu'un AVC :

```
let a = 'Bonjour monsieur le "prof"';
// Bonjour monsieur le "prof"
let b = "Bonjour monsieur le \"prof\"";
// Bonjour monsieur le "prof"
let c = "Bonjour monsieur \"l'enseignant\"";
// Bonjour monsieur "l'enseignant"
```

Ce n'était pas si compliqué au final !

Il existe tout une liste de caractères spéciaux qu'il est possible d'échapper à l'aide du caractère `\` (backspace, ou antislash en français).

Si vous voulez afficher un backslash dans une chaine, vous pouvez l'échapper :

```
let a = 'Youpi \\o/';
```

```
//\o/
let b = '\\\\localhost\\Users\\Bobby';
//\\localhost\\Users\\Bobby
```

Il peut arriver que vous deviez écrire des chaînes de caractères sur plusieurs lignes. Pour cela, vous pouvez utiliser la concaténation par exemple :

```
let houhou = ' ,_ ' +
              '( 8,8)' +
              '( ,, )';
```

Nonobstant cette fonctionnalité remarquable, vous conviendrez du fait que ce n'est pas ultra pratique (et l'exemple pris pour cette chaîne de caractère n'arrange rien...). C'est pour cela qu'ES2016 a apporté un nouveau formalisme bien plus sympathique, à l'aide des backticks ` (ou quotes inversées en français ?) :

```
let houhou = `
  ,_
  ( 8,8)
  ( ,, )`;
```

Pour les hiboux, ce n'est pas flagrant, mais pour indenter du HTML par exemple, la plus-value est incontestable.

Notez qu'en évoquant les chaînes de caractères multi-lignes, nous avons utilisé l'opérateur `+` que vous avez l'habitude d'utiliser en mathématiques. Il s'agit ici d'un opérateur de **concaténation** et non d'addition.

```
const a = 'bulbi';
const b = 'flex';
console.log(a + b);
// bulbiflex
console.log(a + '!' + a + b + '!');
// bulbi ! bulbiflex !
```

Il serait dommage de poursuivre sur ce sujet après un tel exemple. Passons aux structures de données.

Les dates...

Avant de réellement passer aux structures de données, je pense qu'il est important de découvrir le concept de dates en JavaScript. Quiconque ayant déjà travaillé sur un réel projet sait que les dates sont une des meilleures représentations de ce que peut être un cauchemar informatique. Quoi de mieux alors que de vous introduire ce concept dès un premier chapitre me direz-vous ?

Nous allons donc ici découvrir un type de données qui n'est pas considéré comme primitif pour de très bonnes raisons mais que nous allons pratiquement considérer comme tel pour de toutes aussi bonnes raisons.

 Type non primitif traité comme s'il en était un ?!

En effet, à la lecture de la suite du chapitre, vous comprendrez aisément pourquoi nous traitons ici des dates et si je ne vous avais pas dit qu'il s'agissait d'un objet standard de JavaScript et auriez été persuadé qu'il s'agissait d'un type primitif.

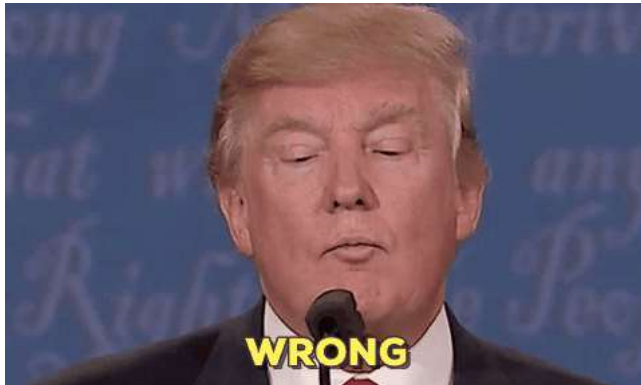
Alors pourquoi n'est ce pas ce que cela semble être ?

La raison est qu'un type est dit primitif si sa définition ne repose pas sur un ou plusieurs autres types du langage. Nous allons par exemple découvrir les objets `Array` qui sont des listes de nombres, chaînes de caractères ou autre : un `Array`, pour exister doit faire intervenir des concepts primitifs et n'est donc pas lui-même un type primitif, alors qu'une chaîne de caractère ne repose sur aucun autre concept. Et c'est là que l'on en arrive aux dates

Créer une date

Nous allons voir une nouvelle manière d'affecter une valeur à une variable. Parce qu'en l'état on ne peut pas faire une chose de ce type

```
const madateEchec = 12/10/2001 // WROOOONG
const madateFail = lundi 12 octobre // WROOOONG
```



En effet, vous sentez à la vue de ces exemples qu'il semble compliqué de trouver un format qui serait évident pour tout le monde. Et en disant tout le monde, je parle de différentes cultures : les pays anglosaxons ne formatent pas les dates comme le font les français, etc. C'est pour ce genre de raisons, entre autre, que nous allons devoir recourir à un autre type d'affectation, ayant ici recours à des constructeurs permettant eux même d'instancier une classe `Date` qui incorpore tout un tas de fonctionnalités et de règles. Et le résultat peut se concrétiser de cette façon :

```
let today = new Date();
console.log(today);
// 2018-12-03T10:38:13.427Z

today = new Date(2018, 11, 25);
console.log("C'est Noel :" + today);
// C'est Noel :Fri Dec 25 2019 00:00:00 GMT+0100 (GMT+01:00)

today = new Date(2018, 11, 25, 12, 0);
console.log("C'est Noel et demi :" + today);
// C'est Noel et demi :Fri Dec 25 2019 12:00:00 GMT+0100 (GMT+01:00)

console.log(new Date('2018/12/25'));
// 2018-12-24T23:00:00.000Z
```

✓ Bonne façon de déclarer des dates

Ici, on constate beaucoup de choses ; Trop de choses... D'abord, nous sommes bien capable de constater que l'on produit des dates. Pour autant, pas mal de choses devraient vous interpeler. Nous n'allons pas entrer dans les détails mais il faut que vous puissiez dès maintenant constater tous ces problèmes.

Que constatez-vous au sujet de ce code et de ses résultats ?

i Réponses possibles

- Plusieurs façons de déclarer des dates (chaîne de caractère ou par X paramètres correspondants à chaque composant)
- Lorsque l'on utilise le constructeur avec une chaîne de caractère, le mois 12 correspond à décembre, lorsque l'on passe par les composants séparés, 12 correspond à Janvier <3
- Construite depuis une chaîne de caractères, la date n'affiche pas la sortie de la même manière que déclarée via le constructeur plus détaillé. (et encore, cela peut même dépendre du navigateur)
- Les timezones... Lorsque l'on produit la date via les composants séparés, cela prend bien compte de notre Timezone (ici GMT+1 puisque nous sommes en France), alors qu'au format chaîne de caractères, la timezone n'est plus la même !

Par ailleurs, lorsque nous avons caractérisé un type primitif comme ne dépendant d'aucun autre type pour se définir. Et il s'avère que si vous tentez de convertir une date en Number, on obtient le nombre de milisecondes écoulées depuis le 1er janvier 1971, à savoir, un nombre.

```
let today = new Date(2018, 12, 25, 12, 0);
console.log(Number(today));
// 1548414000000
```

Nous avons donc commencé à manipuler ici un type de données qui n'est pas primitif mais qui clairement nous sera très vite utile dans nos applications.

Les structures de données

Nous avons vu ce que l'on appelle des types simples, des éléments atomiques. Mais dans la réalité, nous devons souvent manipuler des concepts plus complexes que cela. Prenons le simple exemple d'une liste de notes, ou d'une recette de cuisine : ces deux concepts peuvent respectivement être vus comme une liste de nombres et comme une liste de chaînes de caractères. Ou, lorsque l'on parle d'un humain, on peut vouloir le qualifier par un nom, un prénom, une date de naissance, une taille, etc. et tout ceci pourrait être regroupé au sein d'une structure qui regrouperait à la fois des chaînes de caractères, des nombres et autres. JavaScript offre une panoplie très large de structures de données. Pour ces deux types de besoins (listes et objets complexes), JavaScript offre toute une panoplie d'outils, très complète et assez désagréable à apprendre dans le cadre d'un cours. Je vous renvoie donc à la très complète [documentation de Mozilla Developer Network](#) sur le sujet, vous y apprendrez des choses qui risquent de vous rendre compétent dans le domaine. L'objet de cette section est plus humblement de s'approprier les concepts phares du langage afin que l'appréhension des autres vous semble ensuite bien plus simple.

Les tableaux ou listes

Une liste est une structure permettant de stocker plusieurs instances d'objets, qu'ils soient primitifs, complexes, voire même des structures de données comme des tableaux ou des objets Json que nous allons voir juste après avoir appropris nos tableaux. Pour prouver cela, quelques exemples de tableaux Js :

```
const tab1 = ['Mon', 'tableau', 'de mots', 'lol'];
const tab2 = new Array(10);           // Crée un tableau de 10 cases vides
const tab3 = [1, 2, 3, 'soleil'];     // Un tableau peut accueillir des objets de types distincts

const week = [
  'lundi',
  'mardi',
  'mercredi',
  'jeudi',
  'vendredi',
  'samedi',
  'dimanche',
];
```

☰ Un mot sur ces quelques lignes

Une fois n'est pas coutume : nous avons beaucoup de choses à dire sur les exemples et le formalisme présenté ici.

- Un tableau peut être déclaré via un constructeur, mais pas tout le temps ?!
- des tableaux avec plusieurs types de données dedans ?!
- on déclare des tableaux sur plusieurs lignes ?
- une virgule en trop après dimanche ?

Les exemples que nous venons de manipuler décrivent uniquement des valeurs de type primitif. Pour autant nous avons dit que nous pouvions embarquer des objets (comme des dates par exemple). Mais la force de ces structures de données est de pouvoir même embarquer des structures de données (d'autres tableaux, ou des objets Json).

Les tableaux clé => valeur

Supposons que l'on souhaite représenter une date selon notre propre format, on pourrait vouloir créer un tableau contenant 4 cases : les jours, les numéros de jour dans le mois, les mois et enfin les années. Cela étant, nous nous retrouverions avec un tableau plutôt illisible car en première intention, il serait complexe de comprendre à quoi correspondent ces 4 éléments de notre tableau. Pour ce genre de raisons mais aussi pour des cas bien plus simple, JavaScript propose une approche des tableaux par clé/valeur. On pourrait voir cela comme un tableau de variables : dans notre tableau, les index sont des noms, comme des noms de variables et les valeurs peuvent être de tout type, comme n'importe quelle variable le serait. Dans le cas de notre tableau représentant un "calendrier", nous pourrions donc avoir quelque chose de cet ordre :

```
// Un tableau de 4 cases, avec dans chacune, un autre tableau
let myDate = new Array(4);
myDate['days'] = ['lundi', 'mardi', 'mercredi', '...', 'dimanche'];
myDate['number'] = [1, 2, 3, 4, ..., 31];
myDate['months'] = ['janvier', 'février', 'mars', '...', 'décembre'];
myDate['years'] = [2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020];

// Afficher notre tableau en entier
console.log(myDate);
// Afficher tous les jours de notre grand tableau
console.log(myDate["days"]);
// Afficher toutes les années de notre grand tableau
console.log(myDate["years"]);
// Afficher le jour numéro 0, extrait du tableau 'days' de notre grand tableau
console.log(myDate["days"][0]);
// Afficher le mois numéro 3, extrait du tableau 'months' de notre grand tableau
console.log(myDate["months"][3]);
```

Cette structuration contribue grandement à la lisibilité de notre code, tant lors de la définition du tableau que lors de son utilisation par la suite.

Propriétés et méthodes remarquables

Nous noterons au sujet des propriétés simplement celle qui me semble la plus importante et incontournable : `length` qui permet de récupérer le nombre d'éléments dans notre tableau :

```
const myTab = [1, 2, 0];
console.log(myTab.length)
```

Celq serq tr7s utile lorsque l'on va commencer à parcourir des tableaux par exemple.

Il existe par ailleurs 4 méthodes que nous devons connaître lorsque l'on manipule des tableaux :

- Il est possible d'ajouter un élément en fin de tableau de manière simple :

```
myTab.push(10)
// 1, 2, 0, 10
```

- De la même manière il est facile d'ajouter un élément en début de tableau :

```
myTab.unshift(1234)
// 1234, 1, 2, 0
```

- Comme on peut ajouter un élément en fin de tableau, vous pouvez supprimer un élément en fin de tableau. Cette méthode, en même temps qu'elle supprime le dernier élément du tableau, permet de stocker cette valeur dans une variable :

```
value = myTab.pop()  
// 1234, 1, 2, 0  
console.log(value);  
// 0
```

- Et vous aurez donc deviné qu'il est possible de supprimer un élément en début de tableau en pouvant par la même occasion le stocker dans une variable :

```
value = myTab.shift()  
// 1, 2, 0  
console.log(value);  
// 1234
```

Nous avons donc vu qu'il est simple d'ajouter des éléments en début et en fin de tableau. En revanche, l'insertion au milieu d'un tableau est plus complexe. Vous pouvez notamment utiliser la méthode `splice()` mais c'est en termes de structure de données des choses que vous devriez éviter. Il en est de même pour les suppressions. Il est souvent souhaitable de ne pas

JavaScript Object Notation (Json)

Il existe de nombreux tutoriels présentant ce que peut être Json. Pour autant, il est peut-être intéressant de comprendre sa finalité, son potentiel et certaines de ses subtilités avant de se jeter à corps perdu dans sa mise en oeuvre ou son apprentissage purement théorique technique.

Json par l'exemple

- Nous allons commencer par créer un objet Json représentant un utilisateur d'un site
- Nous allons maintenant représenter un post de blog, mais complet :
 - Le message
 - L'auteur
 - Les métadonnées
 - Les commentaires (et leurs auteurs (et leurs réponses))

Wow ! C'est pas un peu trop ? Clairement ?

Non.

Les objets structurés

Ce type d'objet est contraint par une description qui en est faite a priori. Cela peut sembler être une façon complexe de présenter le concept de classes et c'est tout justement ce que c'est. Cela étant, nous évoquons ici un concept qui mérite que l'on s'y attarde et qui en même temps n'est pas fondamental pour réaliser des sites et applications tout à fait sérieux : c'est pourquoi nous y consacrons le prochain chapitre dans sa totalité.