JavaScript

Variables

Nous déclarons des variables pour stocker des données. Les variables sont var, let ou const.

- ***let***
  Il s'agit d'une déclaration de variable moderne. Dans le système de nommage des variables, let et var se nomment en camelCase.
  Exemple : let userName;
- ***var***
  Il s'agit de la déclaration de variable « old school ». Normalement, on ne l'utilise plus car remplacé par let.
- ***const***
  Il s'agit d'une variable qui, lorsqu'initialisée, ne peut plus être modifiée. Dans le système de nommage des variables, les constantes ont un nom en majuscule.
  Exemple : const COLOR_RED = '#F00';

Les avantages de *var* face à *let* sont :

- Il n'a aucune portée de bloc et donc, il est global, visible à travers les blocs.
- Il est traité dès le début de la fonction, quel que soit l'endroit où il est dans la partie de code.

Types

Il y a 7 types de base en JavaScript :

- ***number*** : pour les numéros de toutes sortes : entier ou décimale.
- ***string*** : pour les chaînes ayant un ou plusieurs caractères.
- ***boolean*** : pour true/false.
- ***null*** : pour les valeurs inconnues (un type autonome qui a une valeur unique).
- ***undefined*** : pour les valeurs non assignées (un type autonome qui a une valeur unique).
- ***object*** : pour des structures de données plus complexes.
- ***symbol*** : pour les indicateurs uniques.

L'opérateur *typeof* nous permet de voir quel type de donnée est stocké dans la variable. Il peut s'écrire sous 2 formes : *typeof x* ou *typeof(x)*. Il retourne une chaîne avec le nom du type mais dans le cas de *null*, il renverra object alors qu'il ne s'agit pas de cela ; c'est une erreur du langage.

En JavaScript, il y a 3 types de citations :

1. Guillemets : `"Hello"`
2. Apostrophes : `'Hello'`
3. Apostrophes inverses : `` `Hello` ``

Contrairement aux 2 autres, l'apostrophe inverse a des fonctionnalités étendues. Il nous permet d'intégrer des variables et des expressions dans une chaîne comme l'exemple plus bas.
${…} va permettre à ce que l'expression à l'intérieur soit évaluée et que le résultat face partie intégrante de la chaîne de caractères.

```
let name = "John";

// embed a variable
alert( `Hello, ${name}!` ); // Hello, John!

// embed an expression
alert( `the result is ${1 + 2}` ); // the result is 3
```

Conversion des types

Il y a 3 conversions de type largement utilisé : la conversion en chaîne, nombre et valeur booléenne. ***ToString***, ***ToNumber*** et ***ToBoolean***.

Les conversions sont faciles à utiliser et comprendre. 2 particularités existent néanmoins :

- undefined est NaN (Not a Number) en tant que nombre et pas 0.
- '0' et des chaînes de caractères comme ' ' sont vrai en booléen.

Dans le cas de *ToNumber*, il suit quelques règles :

Value	Becomes...
`undefined`	`NaN`
`null`	`0`
`true / false`	`1 / 0`
`string`	The string is read "as is", whitespaces from both sides are ignored. An empty string becomes `0`. An error gives `NaN`.

Dans le cas de *ToBoolean*, il suit quelques règles :

Value	Becomes...
`0`, `null`, `undefined`, `NaN`, `""`	`false`
any other value	`true`

Opérateurs

https://javascript.info/operators

Opérateur unaire : ne possède qu'un seul opérande.
Exemple : négation (-).
Opérateur binaire : possède deux opérandes.

Si une expression à plus d'un opérateur, il y a un ordre d'exécution à respecter sur base de la priorité implicite entre les opérateurs. Chaque opérateur possède un numéro de priorité correspondant. Celui avec le plus grand nombre s'exécute d'abord et si la priorité est la même, l'ordre d'exécution est de gauche à droite.

Voici une liste de certains opérateurs (à noter que les opérateurs unaires sont plus élevé que les correspondant binaires) :

Precedence	Name	Sign
...	...	...
16	unary plus	+
16	unary negation	-
14	multiplication	*
14	division	/
13	addition	+
13	subtraction	-
...	...	...
3	assignment	=
...	...	...

D'autres opérateurs

Le reste (%), l'exponentiel (**), l'incrémentation (exemple : i++), la décrémentation (exemple : i--).
Opérateurs de bits : le et (&), le ou (|).

Comparaisons

On peut comparer les valeurs par égalité (==, ===) et/ou supériorité/infériorité (<, >) mais il y a également la comparaison d'inégalités (!=).

Dans le cas des égalités, le double égal compare l'information alors que le triple égal compare l'information et le type.
Concrètement, cela signifie que le triple égal n'effectuera pas de conversion de type d'une chaîne de caractères face à un nombre, contrairement au double égal.

- Les opérateurs de comparaison renvoient toujours une valeur logique.
- Les chaînes de caractères sont comparées lettre par lettre dans l'ordre du dictionnaire (donc, dans l'ordre alphabétique).
- Lorsqu'on compare des valeurs de types différents, ils sont automatiquement convertis en nombre donc attention !
- En comparant null et undefined par ==, ils sont égaux entre eux mais à aucune autre valeur.
- Lorsqu'on utilise des comparaisons comme > et < avec des variables qui peuvent être occasionnellement null ou undefined, il est judicieux d'appliquer une vérification externe afin d'éviter tout problème.

Opérateurs logiques

L'opérateur OR est représenté avec deux symboles de la ligne verticale (||).
Il renvoie vrai tant que l'un des opérandes est vrai.

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```

L'opérateur AND est représenté par deux signes & (&&).
Il renvoie vrai uniquement si les deux opérandes sont vrai sinon faux.

```
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false
```

L'opérateur NOT est représenté par un point d'exclamation (!).
Il accepte un argument et le convertis en booléen avant de retourner sa valeur inverse.

```
alert( !true ); // false
alert( !0 ); // true
```

Si on utilise le double point d'exclamation (!!), il réinverse à nouveau la valeur ; nous obtenons une conversion simple en valeur booléenne. On peut aussi passer par la fonction intégrée `Boolean` pour effectuer la même chose.

```
alert( !!"non-empty string" ); // true
alert( !!null ); // false
alert( Boolean("non-empty string") ); // true
alert( Boolean(null) ); // false
```

Opérateur conditionnel : if/else

L'opérateur if, c'est un peu comme un point d'interrogation.
L'instruction if obtient une condition, il l'évalue en le convertissant en type booléen et si le résultat renvoie `true`, alors il exécute le code.

Il se note comme ceci :

```
if (cond) {
  ...
}
```

Il a droit à un bloc facultatif, « else », qui s'exécute lorsque la condition renvoie `false`.

```
let year = prompt('In which year was ECMAScript-2015 specification published?', '');

if (year < 2015) {
  alert( 'Too early...' );
} else if (year > 2015) {
  alert( 'Too late' );
} else {
  alert( 'Exactly!' );
}
```

Il existe 2 notations pour le if/else : if/else/else if et ?/: . Le système ?/: est utilisé lorsqu'il y a plusieurs conditions mais, il peut très bien être utilisé comme alternative à un if.

```
let age = prompt('age?', 18);

let message = (age < 3) ? 'Hi, baby!' :
  (age < 18) ? 'Hello!' :
  (age < 100) ? 'Greetings!' :
  'What an unusual age!';

alert( message );
```

```
if (age < 3) {
  message = 'Hi, baby!';
} else if (a < 18) {
  message = 'Hello!';
} else if (age < 100) {
  message = 'Greetings!';
} else {
  message = 'What an unusual age!';
```

L'instruction switch

Une déclaration if/else peut être remplacé par switch/case. Il offre un moyen bien plus descriptif pour comparer une même valeur avec plusieurs variantes.

Sa particularité est que le code sera lu au cas par cas. Si le premier cas n'est pas rempli, il testera le second et si le second n'est pas rempli, il passera au troisième et ainsi de suite. Si aucun cas n'est rempli, il exécutera automatiquement le cas `default` s'il est présent.

La présence de `break` dans le code est uniquement pour effectuer une pause, un contrôle.

On peut très bien regrouper plusieurs cas ensemble.

```
switch(x) {
  case 'value1':  // if (x === 'value1')
    ...
    break;
  case 'value2':
  case 'value3':  // if (x === 'value2')
    ...
    break;
  default:
    ...
    break;
}
```

Boucles while et for

Il y a 3 types de boucles :

- ***while***
  La condition est vérifiée avant chaque itération.
- ***do..while***
  La condition est vérifiée après chaque itération.
- ***for(; ;)***
  La condition est vérifiée avant chaque itération et des paramètres supplémentaires sont disponible.

Il vaut mieux éviter de faire une boucle infinie mais dans certains cas, cela peut s'avérer utile. Pour cela, on utilise généralement *while(true)* et on peut l'arrêter avec *break*.
continue permet de pouvoir aller à l'itération suivante plutôt que de continuer sur l'itération actuelle.
break/continue sont tout de même à éviter dans un code.

Fonctions

Une déclaration de fonction ressemble à ceci :

```
function name(parameters, delimited, by, comma) {
/* code */
}
```

- Les valeurs passées à une fonction comme paramètre sont copiées dans ses variables locales.
- Une fonction peut accéder aux variables externes mais ça ne fonctionne que dans un sens. Le code à l'extérieur de la fonction ne peut donc pas voir les variables locales.
- Une fonction peut retourner une valeur.

On affecte généralement un nom facilement compréhensible à toute fonction :

- Le nom de la fonction s'écrit en camelCase, tout comme les variables *let* et *var*.
- Le nom devrait décrire clairement ce que fait la fonction afin de savoir ce qu'il devrait faire et possiblement retourner. Des commentaires pour expliquer plus en détail certaines étapes peut être judicieux.
- Une fonction est une action donc, les noms de fonction sont généralement verbaux.
- Il existe plusieurs préfixes de fonctions bien connue comme *create, show, get, check, …* Les utiliser peut être très judicieux lorsqu'on nomme la fonction.

Tableaux

Les tableaux sont des objets qui permet de stocker et gérer des données.
La propriété *length* nous donne la longueur du tableau.

```
// square brackets (usual)
let arr = [item1, item2...];

// new Array (exceptionally rare)
let arr = new Array(item1, item2...);
```

Différentes opérations sont possibles sur un tableau :

- *push(…items)* permet d'ajouter des éléments à la fin.
- *pop()* retire le dernier élément et l'affiche.
- *shift()* retire le premier élément et l'affiche.
- *unshift(…items)* ajoute un élément au début.

Il est également possible de créer une boucle sur les éléments du tableau :

- `for (let i=0; i<arr.length; i++)` : le plus rapide et compatible tous navigateurs.
- `for (let item of arr)` : la syntaxe moderne des éléments seulement.
- `for (let i in arr)` : à ne jamais utiliser.

Antisèche des méthodes de tableau (anglais)

https://javascript.info/array-methods#summary

- ***To add/remove elements :***
  - `push(...items)` – adds items to the end,
  - `pop()` – extracts an item from the end,
  - `shift()` – extracts an item from the beginning,
  - `unshift(...items)` – adds items to the beginning.
  - `splice(pos, deleteCount, ...items)` – at index `pos` delete `deleteCount` elements and insert `items`.
  - `slice(start, end)` – creates a new array, copies elements from position `start` till `end` (not inclusive) into it.
  - `concat(...items)` – returns a new array: copies all members of the current one and adds `items` to it. If any of `items` is an array, then its elements are taken.
- ***To search among elements :***
  - `indexOf/lastIndexOf(item, pos)` – look for `item` starting from position `pos`, return the index or `-1` if not found.
  - `includes(value)` – returns `true` if the array has `value`, otherwise `false`.
  - `find/filter(func)` – filter elements of through the function, return first/all values that make it return `true`.
  - `findIndex` is like `find`, but returns the index instead of a value.
- ***To transform the array :***
  - `map(func)` – creates a new array from results of calling `func` for every element.
  - `sort(func)` – sorts the array in-place, then returns it.
  - `reverse()` – reverses the array in-place, then returns it.
  - `split/join` – convert a string to array and back.
  - `reduce(func, initial)` – calculate a single value over the array by calling `func` for each element and passing an intermediate result between the calls.
- ***To iterate over elements :***

- `forEach(func)` – calls `func` for every element, does not return anything.

- ***Additionally:***
  - `Array.isArray(arr)` checks `arr` for being an array.

Of all these methods only sort, reverse and splice modify the array itself, the other ones only return a value. These methods are the most used ones, they cover 99% of use cases. But there are few others:

- arr.some(fn)/arr.every(fn) checks the array.
  The function `fn` is called on each element of the array similar to `map`. If any/all results are `true`, returns `true`, otherwise `false`.

- arr.fill(value, start, end) – fills the array with repeating `value` from index `start` to `end`.

- arr.copyWithin(target, start, end) – copies its elements from position `start` till position `end` into *itself*, at position `target` (overwrites existing).

Ressources externes sur : fonctions d'expression et flèches, les objets, résumé

Fonctions d'expression et flèches : https://javascript.info/function-expressions-arrows
Résumé de JavaScript : https://javascript.info/javascript-specials

Les objets : http://javascript.info/object
Les méthodes d'objets : http://javascript.info/object-methods
Convertir un objet en primitive : http://javascript.info/object-toprimitive
Constructeur et opérateur « new » : http://javascript.info/constructor-new