Javascript - Première journée

Qu'est-ce que Javascript ?

C'est un langague de script (!!) qui permet d'ajouter de l'interactivité aux pages WEB.

Si HTML correspond aux gros oeuvre (fondations, murs, plafonds), CSS à la peinture / papier-peint, le Javascript est l'électricité !

Un peu d'histoire ...

Inventeur : Brendan EICH et son équipe en 1995 chez Netscape (un navitateur web)

Nom d'origine : LiveScript

Netscape le renomme Javascript pour profiter d'un effet marketing lié à la popularité de Java MAIS CES LANGAGES N'ONT RIEN A VOIR ENTRE EUX !!!

Microsfot développe alors son langage de script JScript et l'intègre en 1996 à Internet Explorer 3.

En 1997, l'organisation ECMA International crée une version standardisé de Javascript, l'ECMAScript. Aujourd'hui, le Javascript que l'on connait est issu de l'ECMAScript, c'est à dire qu'a chaque fois que l'ECMAScript évolue, les nouveautés finissent par alimenter les navigateurs internet (avec une vitesse de MAJ variable ...).

En soi, le JavaScript n'existe plus, on fait tous de l'ECMAScript, de l'ES. Mais par habitude, on continu à parler de Javascript.

Tout cela, c'est la même chose : JavaScript, JS, ECMAScript, ES.

Comme chaque langage, JS se met à jour. Au début, il s'agissait de grosses mises à jour, puis à partir de 2015, les MAJ sont devenus des itérations annuelles : plus légères car plus régulières, permettant aux navigateurs de pouvoir ajouter les nouveautés au fur et à mesure.

Nom de l'édition	Année de publication	Evènements
ES1	1997	Premiers standards
ES2	1998	Corrections
ES3	1999	Améliorations (String, Error, Number)
ES4	Abandonnée	Aucun accord entre les membres
ES5	2009	Support natif du format JSON
ES6	2015	Nombreuses évolutions qui font du JS un langage moderne
ES7	2016	Ajout de Array.includes()
ES8	2017	Ajout de Object.values() et Object.entries() Ajout de String.padStart() et String.padEnd()
ES9	2018	Améliorations des performances du moteur JS
ES10	2019	Ajout de Array.flat() Ajout de Function.toString()
ES11	2020	Ajout du type BigInt
ES12	2021	Ajout de l'opérateur Nullish `??=`, And Assignment `&&=`, Or Assignment `||=`
ES13	2022	Await operator at the top-level -> en dehors d'une fonction Ajout de la méthode .at(x) aux types indéxables
ESNext	A venir	Nom générique de la future version

On peut faire quoi avec du JS ?

Coté Client

- Assistance de saisie de formulaire
    - Controle et validation
    - Affichage de message d'aide

    - Editeur de texte
- Sauvegarde de données sur le poste local
    - Via les cookie
    - Via une zone disque dédiée (LocalStorage)
- Gestion des nombres, dates et heure.
    - Calculs.
    - Affichage selon le pays
- Animations graphiques
    - Menus
    - Elements esthétique
    - Défilements, diapo, zooms.
    - Graphiques animés et interactifs
    - Cartographies
    - Jeux vidéo
- Saisies de données de haut niveau
    - Tableur type excel
    - Edition de modèle 3D pour impression 3D
    - Création de présentation dynamique type PowerPoint
- Appels asynchrones vers le serveur pour actualiser les données de la page.
    - Cours de la bourse
    - Chat de messagerie
    - Alerte en direct
    - Sauvegarde en temps réel.
    - Mesure d'audience
    - Campagne publicitaire

Si on ajoute NodeJS comme lagage coté server (pas abordé ici) et TypeScript (qui est en gros, l'ESNext disponible aujourd'hui), on comprend que JS reste incontournable, même si parfois méprisé ou relégué à des tâches "simples".

ATTENTION : toutes les fonctionnalités ES ne sont pas implémentés ou supportés par tous les navigateurs. Les utilisateurs utilisent parfois de vieilles machines

Site ressource : Can I use... Support tables for HTML5, CSS3, etc

Ici, TypeScript pourra vous aider.

- JavaScript | MDN -> la base !
- https://www.w3schools.com/ -> parfois un peu plus clair !

La syntaxe

> On va utiliser CodePen pour éxécuter du JS, le temps d'être à l'aise avec la syntaxe. On verra ensuite les endroits où l'on peut exécuter du JS, où l'on peut le stocker et comment le lier avec HTML / CSS ! On utilisera GIT et GitHub aussi ...
>
> TIPS : CodePen est très pratique pour tester rapidement des choses, envoyer du code à d'autres devs pour du débug, etc. (sur les forums notamment).

Déclarer une variable

Le mot-clé **var** est historiquement le moyen de déclarer une variable

```
var age = 5;


console.log(age)

// Cela affiche 5 dans la console

// car la variable age contient le nombre 5.
```

Si on peut encore l'utiliser, son usage est déconseiller car deux autres mots-clés sont venus enrichir JS.

```
let age = 5;


console.log(age)

// Cela vaut toujours 5.


const PI = 3.14;

console.log(PI)

// Cela affiche 3.14 dans la console.
```

Pourquoi **let** et **const** ?

Pour éviter les erreurs.

Imaginons le code suivant :

https://codepen.io/julienpoirierwebdev/pen/zYjBBmR?editors=0011

```
var age = 5;



// Les années passent


age = age + 1;

age = age + 1;

// age est égal à ????

var age = 2;


// que vaut var ?
```

Le mot-clé let empêche de déclarer à nouveau une variable. En soi, déclarer à nouveau une variable avec var n'entraine pas de bug mais ce n'est pas une bonne pratique.

```
let age = 5;

let age = 6;

// Cela va générer une erreur directement.

// Le développeur est alerté du fait

// qu'il a problablement fait n'importe quoi !
```

UNE AUTRE RAISON EXISTE !

Le mot-clé **let** n'est pas soumis au hoisting, alors que **var** l'est. (On peut traduire *hoisting* par "hissage" mais comme les dév doivent faire de l'anglais, on gardera *hoisting*)

Pour résumer simplement : Le hoisting est le fait que les déclarations de variables remontent automatiquement en début de code lors de l'éxécution du script.

En vrai, c'est plus complexe mais on n'a pas besoin d'aller plus loin.

Voici un exemple : https://codepen.io/julienpoirierwebdev/pen/BaxzzGN?editors=0011

```
num = 6;

num + 7;

var num;

/* Ne donne aucune erreur tant que num est déclarée*/
```

Ce code ne génère pas d'erreur, pourtant il n'est pas simple à lire. Avec le mot-clé let, cela n'aurai pas fonctionné. Le code rédigé serait alors plus clair.

Bref, n'utilisez pas **var** mais utilisez **let** !

Ensuite, le mot-clé **const** permet de déclarer une variable qui ne pourra alors plus être modifié !

Un exemple : https://codepen.io/julienpoirierwebdev/pen/MWGeeRv?editors=0011

```
const MYNAME = "Julien"


MYNAME = "Bob" // Oupsi, une erreur !
```

On vérouille une variable que l'on sait être stable !

Tips : on peut déclarer une variable sans l'initialiser !

```
let name; // Ici on déclare


name = "Bob" // Ici on initialise.
```

Un peu de vocabulaire

En JS, on appelle une "instruction" une suite de mot qui génère quelque chose et qui s'exécute de manière isolée.

En général, on considère qu'une instruction se termine par un **;** !

Mais JS est permissif, on peut ne pas le mettre dès qu'on saute une ligne.

Les devs JS mettent parfois le **;** tandis que d'autres non.

```
let name = "Bob" ; let tel; let address, mail ;

// Ici, on a 3 instructions sur une seule ligne.
// Tips : La dernière instruction est une déclaration multiple !
```

Un entrainement sauvage apparait !

---------- 2/3 MIN pour faire l'exercice ----------

https://codepen.io/julienpoirierwebdev/pen/JjvKRdw?editors=1111

Les types de données et le typage en JavaScript.

Le typage en JS est un typage dynamique : cela veut dire qu'une variable peut contenir un type de données puis se voir assigner un type de données, sans que cela ne pose de problème.

D'autres types de langages ne sont pas aussi permissif. Cela a des avantages (il n'y a pas d'erreur qui s'affiche) mais aussi des inconvénients (il n'y a pas d'erreur qui s'affiche).

Il y a 8 types de données en JS !

D'abord, les types primitifs, c'est-à-dire qu'ils ne peuvent contenir qu'une seule chose, en lien avec leur type :

- *Number* - Nombre

Ce sont des données numériques, ce sont des chiffres ! Pas de surprise.

Outre les nombres réguliers, il existe des “valeurs numériques spéciales” qui appartiennent également à ce type: `Infinity`, `-Infinity` et `NaN`.

1. `Infinity` répresente l’Infini ∞ mathématique. C’est une valeur spéciale qui est plus grande que n’importe quel nombre.

   Nous pouvons l’obtenir à la suite d’une division par zéro :

   ```
   alert( 1 / 0 ); // Infinity
   ```

   Ou mentionnez-le simplement dans le code directement :

   ```
   alert( Infinity ); // Infinity
   ```

2. `NaN` représente une erreur de calcul. C’est le résultat d’une opération mathématique incorrecte ou non définie, par exemple :

   ```
   alert( "pas un nombre" / 2 ); // NaN, une telle division est erronée
   ```

   `NaN` est contagieux. Toute autre opération sur `NaN` donnerait un `NaN` :

   ```
   alert( NaN + 1 ); // NaN alert( 3 * NaN ); // NaN alert( "not a number" / 2 - 1 ); // NaN
   ```

   Donc, s’il y a `NaN` quelque part dans une expression mathématique, il se propage à l’ensemble du résultat (il n’y a qu’une seule exception : `NaN ** 0` vaut `1`).

- *BigInt* - Les très grands nombres

Le type *number* a une limite en taille. ES2020 a ajouté ce type. Sachez qu'il existe et qu'il suffit d'ajouter un *n* à la fin d'un nombre pour le transformer en *BigInt*

- *String* - Texte

Pas de surprise non plus. C'est du texte. Il peut contenir des chiffres mais qui ne sont pas considéré comme des nombres (??? quoi ???).

Une chaîne de caractères en JavaScript doit être entre guillemets.

```
let str = "Hello"; let str2 = 'Single quotes are ok too'; let phrase = `can embed another ${str}`;
```

En JavaScript, il existe 3 types de guillemets.

1. Double quotes: `"Hello"`.
2. Single quotes: `'Hello'`.
3. Backticks: `` `Hello` ``.

Les guillemets simples et doubles sont des guillemets “simples”. Il n’y a pratiquement pas de différence entre eux en JavaScript.

Les backticks sont des guillemets “à fonctionnalité étendue”. Ils nous permettent d’intégrer des variables et des expressions dans une chaîne en les encapsulant dans `${…}`, par exemple :

```
let name = "John";

// une variable encapsulée

alert( `Hello, *${**name}*!` ); // Hello, John!

// une expression encapulée
alert( `the result is *${**1 + 2}*` ); // le résultat est 3
```

L’expression à l’intérieur de `${…}` est évaluée et le résultat devient une partie de la chaîne. On peut y mettre n’importe quoi : une variable comme `name` ou une expression arithmétique comme `1 + 2` ou quelque chose de plus complexe.

Veuillez noter que cela ne peut être fait que dans les backticks. Les autres guillemets ne permettent pas une telle intégration !

```
alert( "the result is ${1 + 2}" ); // le résultat est ${1 + 2} (les doubles quotes ne font rien)`
```

- *Boolean* - Booléen

Ce type de données ne peut avoir que deux valeurs : true / false. Vrai ou faux. C'est le type de données le plus pure, qu'il ne faut pas hésiter à utiliser car il est très pratique. C'est un peu comme un voyant : c'est allumé ou éteint, cela signifie quelque chose dans votre code.

Votre personnage a un nombre de vie ? Stockez cette valeur numérique dans une variable. Si le nombre de vie du personnage est inférieur à 1, alors le jeu est terminé. On peut stocker dans une variable game_over si le jeu est terminé ou non. True or False. Et décidé de faire des choses dans le code si le jeu est terminé (ne plus afficher un écran par exemple).

> C'est BOOLE en 1854 qui, dans son algèble, a défini une variable ne pouvant contenir que deux états, un siècle avant l'apparition de l'informatique.

- La valeur *null* - Vide, Rien, Inconnu

La valeur spéciale `null` n'appartient à aucun type de ceux décrits ci-dessus.

Il forme un type bien distinct qui ne contient que la valeur `null` :

```
let age = null;
```

En JavaScript, `null` n'est pas une "référence à un objet non existant" ou un "pointeur nul" comme dans d'autres langages.

C'est juste une valeur spéciale qui a le sens de "rien", "vide" ou "valeur inconnue".

Le code ci-dessus indique que l'`age` est inconnu.

- *Undefined* - Pas attribué !

Ce type est celui d'une variable qui aurait été déclaré mais pas encore initialisé. Il revient lorsque vous tentez d'appeler la propriété d'un objet qui n'existe pas (encore ?)

- *Object* - Objet

Un peu plus complexe et présent partout dans JS. Ce n'est pas un type primitif car un objet peut stocker une collection de données de types différents.

C'est un type générique qui défini un structure de données complexes. Les objets peuvent avoir des propriétés ou des méthodes.

On va voir cela plus en profondeur plus tard.

La structure de base d'un objet est une liste de propriétés contenu entre accolades. Une propriété est composé d'une clé et d'une valeur.

```
let my_object = {

    nom: "Bob",

    is_usefull: false

}
```

Une propriété d'un objet peut contenir tout type de données : des *strings*, des *numbers*, des objets, ...

```
let voiture = {

    couleur: "rouge",

    nb_km: 20000,

    propriétaire: {

        name: "Bob",

        age : 5

    }

}
```

- *Symbol* - Un identifiant unique pour vos objets

Personnellement, je n'ai jamais utilisé ce type et il a été crée "recemment".

Il y a des subtilités

- *NaN - Not a Number* dont le type est ... number.
- Les *Arrays*, des listes, qui sont considérés comme étant de type *Object*, mais qui ont des méthodes propres.

L'opérateur *typeof* permet de tester le type d'une variable / d'une expression.

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Operators/typeof

Et les conversions de type dans tout ça ?

La plupart du temps, les opérateurs et les fonctions convertissent automatiquement les valeurs qui leur sont attribuées dans le bon type.

Par exemple, `alert` convertit automatiquement toute valeur en chaîne de caractères pour l'afficher. Les opérations mathématiques convertissent les valeurs en nombres.

Il y a aussi des cas où nous devons convertir explicitement une valeur pour corriger les choses.

```
// Pour convertir en String
let number = 1;
let string = String(nu// Onmber)
console.log(typeof string) // string


// ------------------

let str = "123";
alert(typeof str); // string

let num = Number(str); // devient un nombre 123
alert(typeof num); // nombre


// On ne peut pas convertir en nombre des lettres.
// On peut toutefois convertir true et false en nombre ...

alert( Number(true) );        // 1
alert( Number(false) );       // 0


// Pour tester l'existence du contenu d'une variable,
// on peut la convertir en booléen

alert( Boolean(1) ); // true
alert( Boolean(0) ); // false

alert( Boolean("hello") ); // true
alert( Boolean("") ); // false


// Attention aux pièges :

alert( Boolean("0") ); // ?


let number = 1
let letter = "1"


console.log(number + letter) // ?


console.log(letter + letter) // ?

console.log(letter *3) // ?
```

Calculer en Javascript !

En JS, on peut faire beaucoup de math ! ou plutôt, JS fait beaucoup de Math pour nous !

On peut faire beaucoup de choses simples.

Les opérations mathématiques suivantes sont supportées :

- Addition `+`,
- Soustraction `-`,
- Multiplication `*`,
- Division `/`,
- Reste `%`,
- Exponentiation `**`.

```
let ps5 = 500;


let pack_of_games = 120;


let percent_of_reduc = 5;


let total = ps5 + pack_of_games;


let saving = total * ( 100 - percent_of_reduc) / 100;



console.log("Votre panier s'élève à " + total + " €, en prenant compte la réduction de
" + percent_of_reduc + "% !")
```

Quand on additionne plusieurs string ou une string et d'autres éléments, on créer une nouvelle string.

Cela s'appelle de la concaténation.

```
let name = "Bob";


console.log("Hello ${name} !")
```

Si la concatenation nous semble fastidieuse, on peut faire de l'interpolation. Il s'agit d'insérer du Javascript directement dans une string, avec l'élément ${} !

Tips : c'est un rappel à php, où $ se met avant les variables et permet directement l'interpolation.

Un peu de vocabulaire : “unaire”, “binaire”, “opérande”

https://fr.javascript.info/operators#termes-unaire-binaire-operande)

Avant de continuer, saisissons la terminologie commune.

- Un opérande est ce à quoi les opérateurs sont appliqués. Par exemple, dans la multiplication `5 * 2`, il y a deux opérandes: l’opérande gauche est `5` et l’opérande droit est `2`. Parfois, les gens disent “arguments” au lieu de “opérandes”.
- Un opérateur est unaire s’il a un seul opérande. Par exemple, la négation unaire `-` inverse le signe du nombre :

```
let x = 1;

*x = -x;*
alert( x ); // -1, le moins unaire a été appliqué`
```

- Un opérateur est *binaire* s’il a deux opérandes. La même négation existe également dans la forme binaire :

```
let x = 1, y = 3;
alert( y - x ); // 2, le moins binaire soustrait des valeurs`
```

D’un point de vue formel, dans les exemples ci-dessus, nous avons deux opérateurs différents qui partagent le même symbole : l’opérateur de négation, un opérateur unaire qui inverse le signe, et l’opérateur de soustraction, un opérateur binaire qui soustrait un nombre d’un autre.

Un entrainement sauvage apparait !

---------- 10 MIN pour faire l'exercice ----------

https://codepen.io/julienpoirierwebdev/pen/OJZXRNN

Le modulo %

L'opérateur modulo permet de connaître le reste d'une division.

```
let prix = 25;

let ten_bucks_value = 10;


let modulo_10_bucks = prix % 10;



let number_of_ten_bucks = prix - modulo_of_10_bucks / ten_bucks_value ;
```

Les subtilités !

Déja, la "précédence" s'applique, comme en mathématique : une multiplication ou une division s'exécute avant les additions et les soustractions, sauf si des parenthèses isolent des opérations.

Lorsque l'on initialise une variable, le symbole = est un opérateur ! Cela veut dire que s'il est utilisé dans une expression, il a aussi ce rôle d'initialisation.

Un exemple est plus parlant :

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3
alert( c ); // 0
```

ATTENTION : Ce n'est ni lisible, ni recommandé, ni une bonne pratique.

On peut aussi faire des affectations chainées : ne le faites pas mais sachez que parfois, certains le font :

```
let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

Ici, le dernier élément à droite est affecté comme valeur aux élements à gauche.

La modification "sur place"

```
let n = 2;
n = n + 5;
n = n * 2;
```

Cette notation peut être raccourcie en utilisant les opérateurs `+=` et `*=` :

```
let n = 2;
n += 5; // maintenant n = 7 (identique à n = n + 5)
n *= 2; // maintenant n = 14 (identique à n = n * 2)

alert( n ); // 14
```

Il existe de opérateurs de “modification et assignation” courts pour tous les opérateurs arithmétiques et binaires: `/=`, `-=` etc.

Ces opérateurs ont la même précédence qu’une affectation normale. Ils s’exécutent donc après la plupart des autres calculs :

```
let n = 2;

n *= 3 + 5;

alert( n ); // 16 (partie droite évaluée en premier, identique à n *= 8)`
```

Incrémentation et décrémentation

L’augmentation ou la diminution d’un nombre par `1` compte parmi les opérations numériques les plus courantes.

Il y a donc des opérateurs spéciaux pour cela :

- **Incrémentation** `++` augmente une variable de 1 :

  ```
  let counter = 2;
  counter++; // fonctionne de la même manière que counter = counter + 1, mais c'est
  plus court
  alert( counter ); // 3
  ```

- **Décrémentation** `--` diminue une variable de 1 :

```
let counter = 2;
counter--; // fonctionne de la même manière que counter = counter - 1, mais c'est plus court
alert( counter ); // 1
```

Les opérateurs binaires

Sachez qu'ils existent et servent surtout lors des calculs bas niveaux (bit à bit par exemple) ou en crypto.

https://fr.javascript.info/operators#operateurs-binaires

Expressions et opérateurs - JavaScript | MDN

Un exercice sauvage apparait !

https://fr.javascript.info/operators#les-conversions-de-type

Les blocs d'instruction, les tests conditionnels, les opérateurs ! La base de l'algorithmie !

Les comparateurs ... !

Il y a de nombreux opérateurs de comparaison que nous connaissons des mathématiques :

- Plus grand/petit que : `a > b`, `a < b`.
- Plus grand/petit ou égal à : `a >= b`, `a <= b`.
- Égalité : `a == b` (veuillez noter le signe de la double égalité `==` signifie un test d'égalité. Un seul symbole `a = b` signifierait une affectation).
- Pas égal : en maths la notation est `≠`, mais en JavaScript elle est écrite comme une assignation avec un signe d'exclamation : `a != b`.

Le résultat d'une comparaison est toujours un booléen ! *true* or *false*

Egalité stricte ou égalité simple ?

Observez le code suivant :

```
let two_as_number = 2;
let two_as_string = "2";

console.log(two_as_number == two_as_string) // ?
console.log(two_as_number === two_as_string) // ?

let another_number = 10;

let result1 = two_as_number + another_number
let result2 = two_as_string + another_number

console.log(result1) // ?
console.log(result2) // ?
```

- Les opérateurs de comparaison renvoient une valeur logique.
- Les chaînes de caractères sont comparées lettre par lettre dans l'ordre "dictionnaire".
- Lorsque des valeurs de différents types sont comparées, elles sont converties en nombres (à l'exclusion d'un contrôle d'égalité strict).
- Les valeurs `null` et `undefined` sont égales `==` et ne correspondent à aucune autre valeur.
- Soyez prudent lorsque vous utilisez des comparaisons telles que `>` ou `<` avec des variables pouvant parfois être `null/undefined`. Faire une vérification séparée pour `null/undefined` est une bonne idée.

TIPS : Privilégier le contrôle d'égalité stricte pour éviter les comportements non attendus du code. Il vaut mieux débuguer au début plutôt qu'après !

Du concret et de la condition !

JS s'exécute de haut en bas mais on peut l'empêcher d'exécuter du code si un test n'est pas réussi.

```
let age = 5;


if(age >= 18) {
    console.log("Vous êtes majeur !")
} else {
    console.log("Vous êtes mineur")
}
```

```
let age = 5;


if(age < 13) {
    console.log("Vous n'avez pas le droit d'être sur les réseaux sociaux ! Stoppez immédiatement cette danse tiktok")
} else if(age >= 18) {
    console.log("Vous êtes majeur, vous êtes libre mais pénalement responsable ! Aie !")
} else {
    console.log("Aller, ok pour tiktok ... ")
}
```

```
let class = "";


switch(class) {

}
```

exo : https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-4.php

https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-6.php

https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-8.php

https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-10.php

https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-11.php

https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-22.php

https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-25.php

https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-38.php

https://www.w3resource.com/javascript-exercises/javascript-basic-exercise-48.php

https://www.webtips.dev/how-i-made-a-snake-game-out-of-checkboxes

https://codepen.io/CaioPaiola/pen/nojJmQ

https://github.com/Beat0154/easiest-game-ever

Les fonctions et les boucles ! Pour gagner du temps ...