

1

Programmation JavaScript



► Développement Digital

mohamed@goumih.com

Sommaire:

2

- ▶ **Partie 1:Présentation de JavaScript & outils de travail**
- ▶ **Partie 2:les fondamentaux de JavaScript**
- ▶ **Partie 3:Les objets**
- ▶ **Partie 4:Nombres et Strings**
- ▶ **Partie 5:Les Arrays**
- ▶ **Partie 6:Les itérables**
- ▶ **Partie 7:Stockage interne**
- ▶ **Partie 8:Manipulation du DOM**
- ▶ **Partie 9: Les expressions régulières**
- ▶ **Partie 10:Orienté Objet**
- ▶ **Partie 11:JavaScript Asynchrone**
- ▶ **Partie 12:Jquery**
- ▶ **Partie 13:Allez plus loin**

Dans chaque partie il y a des exercices pratiques(**EP**) et des travaux pratiques(**TP**)

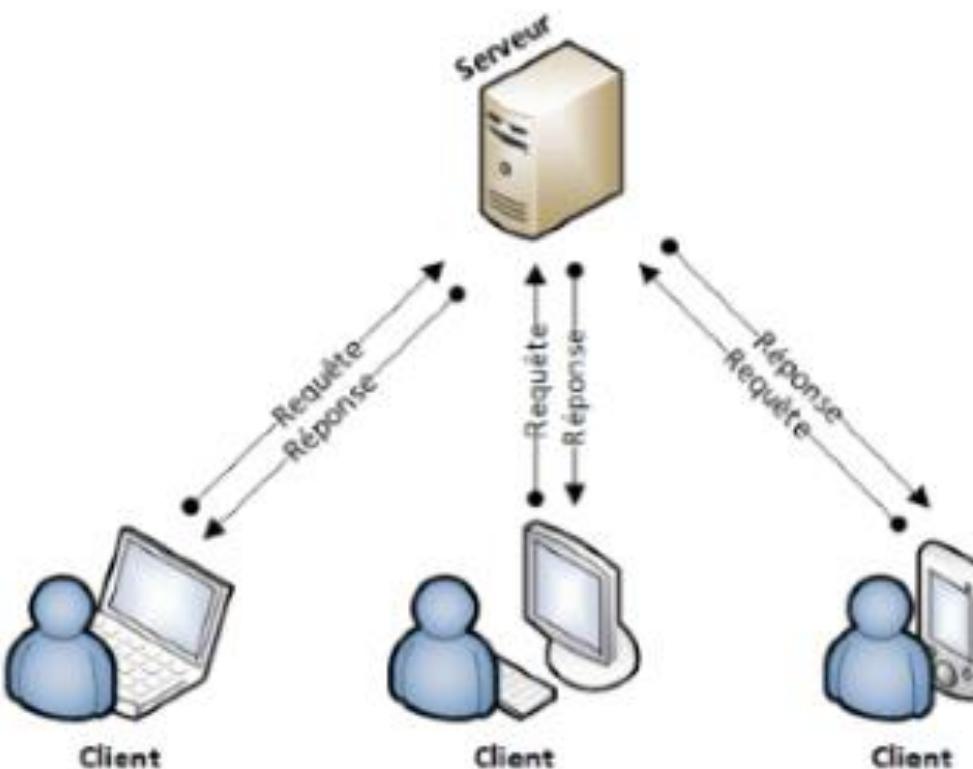
Partie 1:

**Présentation de
JavaScript
et outils de travail**

Architecture client-serveur

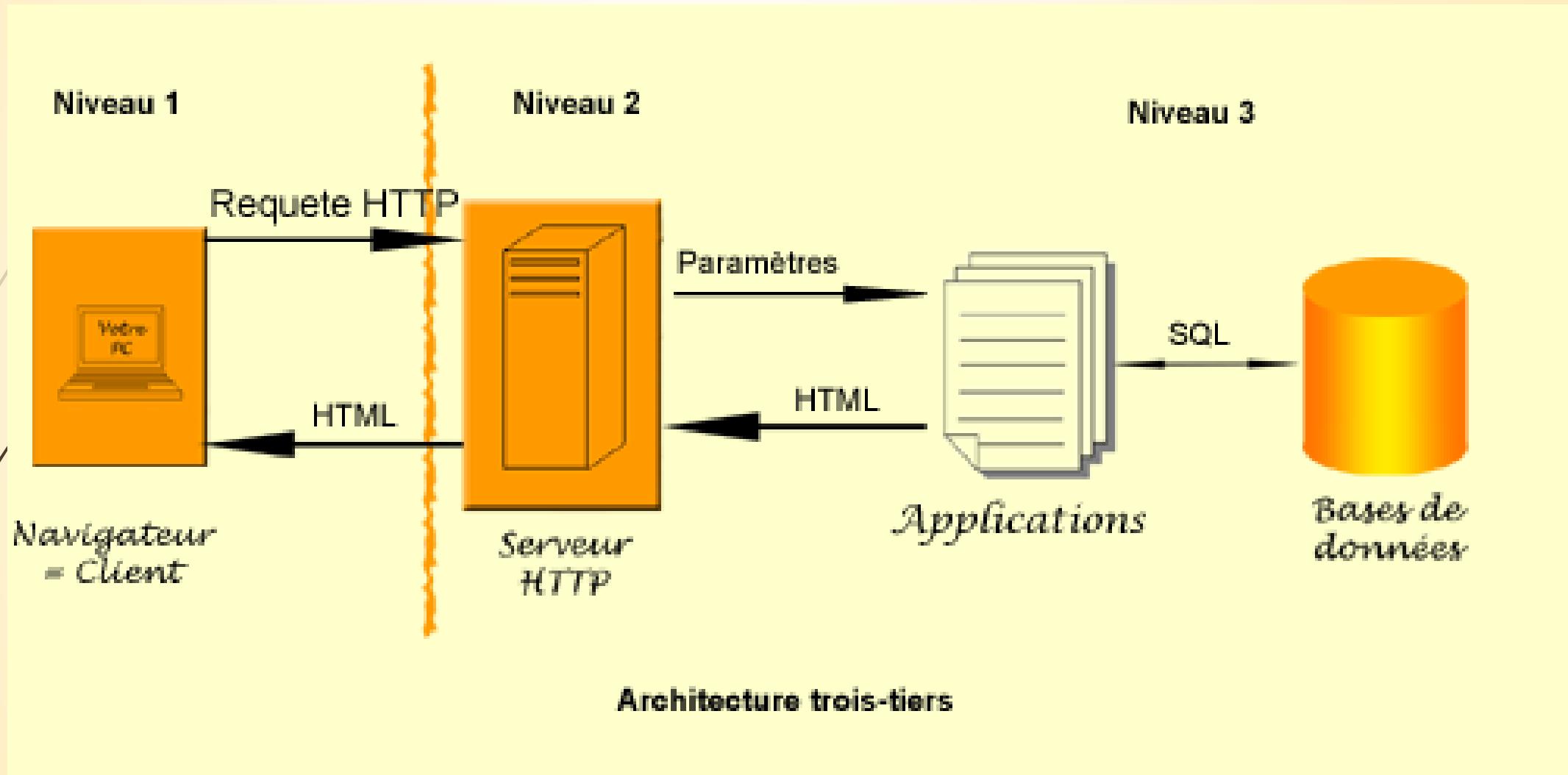
4

- Une architecture client-serveur représente l'environnement dans lequel des applications de machines clientes communiquent avec des applications de machines de type serveurs.
- L'exemple classique est le navigateur Web d'un client qui demande (on parle de "requête") le contenu d'une page Web à un serveur Web qui lui renvoie le résultat (on parle de "réponse").



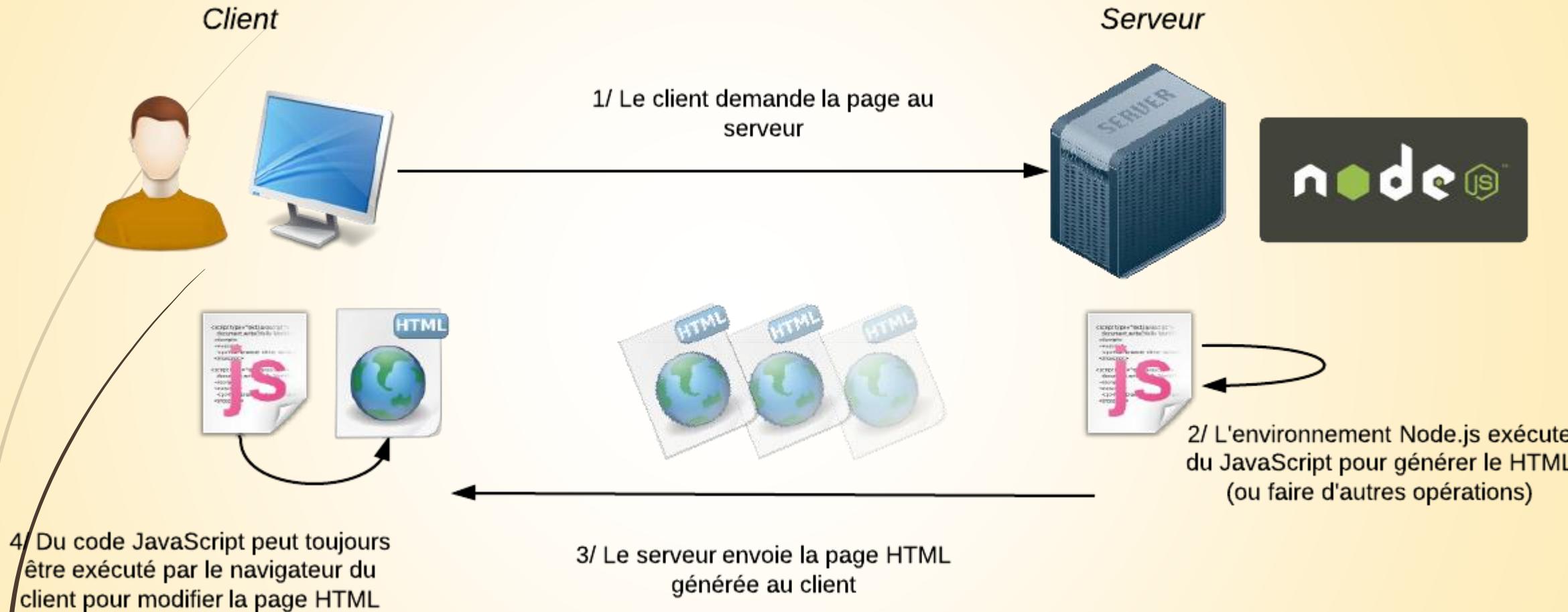
Architecture Client/serveur

5



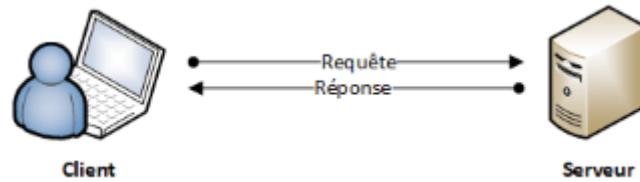
Architecture Client/serveur

6

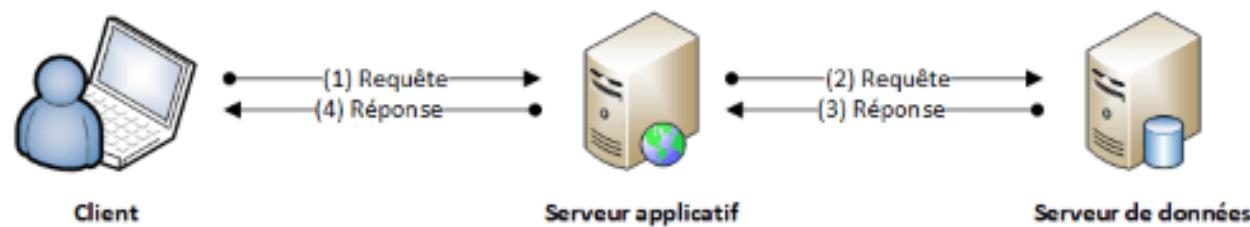


Les types d'architecture client-serveur

- Si toutes les ressources nécessaires sont présentes sur un seul serveur, on parle d'architecture à deux niveaux ou 2 tiers (1 client + 1 serveur).



- Si certaines ressources sont présentes sur un deuxième serveur (par exemple des bases de données), on parle d'architecture à trois niveaux ou 3 tiers (1 client interroge le premier serveur qui lui-même interroge le deuxième serveur).
- Au delà de 3 acteurs, on parle d'architecture à n tiers.



Présentation de Javascript

8

- ▶ **JavaScript** est un [langage de programmation](#) de [scripts](#) principalement employé dans les [pages web](#) interactives et à ce titre est une partie essentielle des [applications web](#). Avec les langages [HTML](#) et [CSS](#), JavaScript est au cœur des langages utilisés par les [développeurs web](#). Une grande majorité des [sites web](#) l'utilisent, et la majorité des [navigateurs web](#) disposent d'un [moteur JavaScript](#) pour l'[interpréter](#) .
- ▶ JavaScript est aussi employé pour les [serveurs Web](#) avec l'utilisation (par exemple) de [Node.js..](#)
- ▶ JavaScript a été créé en 1995 par [Brendan Eich](#) et intégré au [navigateur web Netscape Navigator](#) 2.0. L'implémentation concurrente de JavaScript par [Microsoft](#) dans [Internet Explorer](#) jusqu'à sa version 9 se nommait [JScript](#), tandis que celle d'[Adobe Systems](#) se nommait [ActionScript](#). JavaScript a été standardisé sous le nom d'[ECMAScript](#) en juin 1997 par [Ecma International](#) dans le standard ECMA-262. La version en vigueur de ce standard depuis juin 2021 est la 12^e édition.
- ▶ C'est un langage [orienté objet à prototype](#): les bases du langage et ses principales [interfaces](#) sont fournies par des [objets](#). Cependant, à la différence d'un langage orienté objets, les objets de base ne sont pas des [instances de classes](#). En outre, les [fonctions](#) sont des [objets de première classe](#). Le langage supporte le [paradigme objet, impératif et fonctionnel](#).
- ▶ JavaScript est le langage possédant le plus large [écosystème](#) grâce à son gestionnaire de dépendances [npm..](#)

Présentation de Javascript

9

- ▶ *JavaScript* a été initialement créé pour « rendre les pages Web vivantes ».
- ▶ Les programmes dans ce langage sont appelés *scripts*. Ils peuvent être écrits directement dans le code HTML d'une page Web et s'exécuter automatiquement au fur et à mesure du chargement de la page.
- ▶ Les scripts sont fournis et exécutés en texte brut. Ils n'ont pas besoin d'une préparation ou d'une compilation spéciale pour fonctionner.
- ▶ Dans cet aspect, *JavaScript* est très différent d'un autre langage appelé *Java*.

Pourquoi s'appelle-t-il JavaScript ?

10

- ▶ Lorsque JavaScript a été créé, il avait initialement un autre nom: « LiveScript ». Mais Java était très populaire à cette époque, il a donc été décidé que le positionnement d'un nouveau langage en tant que « frère cadet » de Java aiderait.
- ▶ Mais au fur et à mesure de son évolution, JavaScript est devenu un langage entièrement indépendant avec sa propre spécification appelée ECMAScript, et maintenant il n'a aucun rapport avec Java.
- ▶ Aujourd'hui, JavaScript peut s'exécuter non seulement dans le navigateur, mais aussi sur le serveur, ou en fait sur n'importe quel appareil doté d'un programme spécial appelé moteur JavaScript
- ▶ Le navigateur dispose d'un moteur embarqué parfois appelé « machine virtuelle JavaScript ».
- ▶ Différents moteurs ont des « noms de code » différents. Par exemple:
 - V8 – dans Chrome, Opera et Edge.
 - SpiderMonkey – dans Firefox.
 - ... Il existe d'autres noms de code comme « Chakra » pour IE, « JavaScriptCore », « Nitro » et « SquirrelFish » pour Safari, etc.

Comment fonctionnent les moteurs ?

- ▶ Les moteurs sont compliqués. Mais les bases sont faciles.
 1. Le moteur (intégré s'il s'agit d'un navigateur) lit (« analyse ») le script.
 2. Ensuite, il convertit (« compile ») le script en langage machine.
 3. Et puis le code machine s'exécute, assez rapidement.
- ▶ Le moteur applique des optimisations à chaque étape du processus. Il observe même le script compilé pendant son exécution, analyse les données qui le traversent et optimise davantage le code machine en fonction de ces connaissances.

Que peut faire JavaScript dans le navigateur ?

- ▶ JavaScript moderne est un langage de programmation « sûr ». Il ne fournit pas d'accès de bas niveau à la mémoire ou au processeur, car il a été initialement créé pour les navigateurs qui n'en ont pas besoin.
- ▶ Les capacités de JavaScript dépendent grandement de l'environnement dans lequel il s'exécute. Par exemple, **Node.js** prend en charge des fonctions qui permettent à JavaScript de lire/écrire des fichiers arbitraires, d'effectuer des requêtes réseau, etc.
- ▶ JavaScript dans le navigateur peut faire tout ce qui concerne la manipulation de pages Web, l'interaction avec l'utilisateur et le serveur Web.
- ▶ Par exemple, JavaScript dans le navigateur est capable de :
 - Ajoutez du nouveau code HTML à la page, modifiez le contenu existant, modifiez les styles.
 - Réagissez aux actions de l'utilisateur, exécutez sur les clics de souris, les mouvements du pointeur, les pressions sur les touches.
 - Envoyer des requêtes sur le réseau à des serveurs distants, télécharger et télécharger des fichiers (technologies AJAX..).
 - Obtenez et définissez des cookies, posez des questions au visiteur, affichez des messages.
 - Mémorisez les données côté client (« stockage local »).

Que ne peut pas faire JavaScript dans le navigateur

mohamed@goumih.com

13

Les capacités de JavaScript dans le navigateur sont limitées pour la sécurité de l'utilisateur. L'objectif est d'empêcher une page Web malveillante d'accéder à des informations privées ou de nuire aux données de l'utilisateur.

Voici des exemples de telles restrictions :

- JavaScript sur une page Web ne peut pas lire/écrire des fichiers arbitraires sur le disque dur, les copier ou exécuter des programmes. Il n'a pas d'accès direct aux fonctions du système d'exploitation.

Les navigateurs modernes lui permettent de fonctionner avec des fichiers, mais l'accès est limité et n'est fourni que si l'utilisateur effectue certaines actions, comme « déposer » un fichier dans une fenêtre de navigateur ou le sélectionner via une balise <input>. Il existe des moyens d'interagir avec la caméra/microphone et d'autres appareils, mais ils nécessitent l'autorisation explicite de l'utilisateur. Ainsi, une page compatible JavaScript peut ne pas activer surnoiselement une caméra Web

- Les différents onglets/fenêtres ne se connaissent généralement pas. Parfois, ils le font, par exemple lorsqu'une fenêtre utilise JavaScript pour ouvrir l'autre. Mais même dans ce cas, JavaScript d'une page peut ne pas accéder à l'autre s'ils proviennent de sites différents (d'un domaine, d'un protocole ou d'un port différent).

C'est ce qu'on appelle la « politique de même origine ». Pour contourner ce problème, *les deux pages* doivent accepter l'échange de données et contenir un code JavaScript spécial qui le gère. Nous couvrirons cela dans le tutoriel.

- Cette limitation est, encore une fois, pour la sécurité de l'utilisateur. Une page <http://anysite.com> qu'un utilisateur a ouverte ne doit pas pouvoir accéder à un autre onglet du navigateur avec l'URL <http://gmail.com> et voler des informations à partir de là.
- JavaScript peut facilement communiquer sur le net avec le serveur d'où provient la page actuelle. Mais sa capacité à recevoir des données d'autres sites / domaines est paralysée. Bien que possible, cela nécessite un accord explicite (exprimé en en-têtes HTTP) du côté distant. Encore une fois, c'est une limitation de sécurité.

Qu'est-ce qui rend JavaScript unique ?

- ▶ Il y a au moins *trois* grandes choses à propos de JavaScript :
 - Intégration complète avec HTML/CSS.
 - Les choses simples sont faites simplement.
 - Pris en charge par tous les principaux navigateurs et activé par défaut.
- ▶ JavaScript est la seule technologie de navigateur qui combine ces trois choses.
- ▶ C'est ce qui rend JavaScript unique. C'est pourquoi c'est l'outil le plus répandu pour créer des interfaces de navigateur.
- ▶ Cela dit, JavaScript permet également de créer des serveurs, des applications mobiles, etc.

Langages « sur » JavaScript (over)

15

- ▶ La syntaxe de JavaScript ne convient pas aux besoins de tout le monde. Différentes personnes veulent des fonctionnalités différentes.
- ▶ C'est à prévoir, car les projets et les exigences sont différents pour tout le monde.
- ▶ Ainsi, récemment, une pléthore de nouveaux langages sont apparus, qui sont *transpilés (convertis)* en JavaScript avant de s'exécuter dans le navigateur.
- ▶ Les outils modernes rendent la transpilation très rapide et transparente, permettant aux développeurs de coder dans un autre langage et de le convertir automatiquement « sous le capot ».
- ▶ Exemples de ces langues:
 - CoffeeScript est un « sucre syntaxique » pour JavaScript. Il introduit une syntaxe plus courte, ce qui nous permet d'écrire du code plus clair et plus précis. Habituellement, les développeurs Ruby l'aiment.
 - TypeScript se concentre sur l'ajout d'un « typage de données strict » pour simplifier le développement et la prise en charge de systèmes complexes. Il est développé par Microsoft.
 - Flow ajoute également le typage de données, mais d'une manière différente. Développé par Facebook.
 - Dart est un langage autonome qui possède son propre moteur qui s'exécute dans des environnements autres que les navigateurs (comme les applications mobiles), mais qui peut également être transposé en JavaScript. Développé par Google.
 - Brython est un transpiler Python en JavaScript qui permet l'écriture d'applications en Python pur sans JavaScript.
 - Kotlin est un langage de programmation moderne, concis et sûr qui peut cibler le navigateur ou le nœud.
- ▶ Il y en a d'autres. Bien sûr, même si nous utilisons l'un des langages transpilés, nous devrions également connaître JavaScript pour vraiment comprendre ce que nous faisons.

Résumé :

JavaScript a été initialement créé en tant que langage réservé aux navigateurs, mais il est maintenant utilisé dans de nombreux autres environnements.

Aujourd’hui, JavaScript occupe une position unique en tant que langage de navigateur le plus largement adopté, entièrement intégré à HTML / CSS.

Il existe de nombreux langages qui sont « transpilés » en JavaScript et fournissent certaines fonctionnalités. Il est recommandé d’y jeter un coup d’œil, au moins brièvement, après avoir maîtrisé JavaScript.

Documentation utile

17

- ▶ ECMA-262 est le nom officiel de la norme. ECMAScript est le nom officiel de la langue :
- ▶ Documentation officielle modifiée chaque 6 mois:
- ▶ <https://tc39.es/ecma262/#sec-intro>
- ▶ Pour en savoir plus sur les nouvelles fonctionnalités de pointe, y compris celles qui sont « presque standard » (appelées « étape 3 »), voir les propositions à <https://github.com/tc39/proposals>

Pour s'assurer que le code que vous écrivez fonctionne sur tous les navigateurs, il est courant de transpiler votre Code Javascript. Transpiling traduit votre code JavaScript d'une version à une précédente Version ES. Cela vous permet à son tour d'utiliser les dernières fonctionnalités ES qui n'ont pas été mis en œuvre uniformément sur tous les navigateurs. Babel est un transpileur commun <https://www.babeljs.io>).

Tables de compatibilité

Pour voir leur prise en charge parmi les moteurs basés sur un navigateur et d'autres moteurs, voir:

- <http://caniuse.com> – tableaux de support par fonctionnalité, par exemple pour voir quels moteurs prennent en charge les fonctions de cryptographie modernes: <http://caniuse.com/#feat=cryptography>.
- <https://kangax.github.io/compat-table> – un tableau avec des fonctionnalités de langue et des moteurs qui les prennent en charge ou ne prennent pas en charge.

Documentation utile

18

- **MDN (Mozilla) JavaScript Reference** est le manuel principal avec des exemples et d'autres informations.
- C'est génial d'obtenir des informations détaillées sur les fonctions linguistiques individuelles, les méthodes, etc. On peut le trouver à <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>.

Cependant, il est souvent préférable d'utiliser une recherche sur Internet à la place. Il suffit d'utiliser « MDN [terme] » dans la requête, par exemple <https://google.com/search?q=MDN+parseInt> pour rechercher la fonction parseInt.
Pour plus :
<https://www.w3schools.com/js>

Toutes ces ressources sont utiles dans le développement de la vie réelle, car elles contiennent des informations précieuses sur les détails de la langue, leur support, etc. **S'il vous plaît rappelez-vous bien ses deux pages pour les cas où vous avez besoin d'informations détaillées sur une fonctionnalité particulière.**

Environnement de travail & outils

► IDE

- Le terme IDE(Integrated Development Environment) fait référence à un éditeur puissant avec de nombreuses fonctionnalités qui fonctionne généralement sur un « projet entier ». Comme son nom l'indique, il ne s'agit pas seulement d'un éditeur, mais d'un « environnement de développement » à grande échelle.
- Un IDE charge le projet (qui peut être de nombreux fichiers), permet la navigation entre les fichiers, fournit une autocomplétion basée sur l'ensemble du projet (pas seulement le fichier ouvert) et s'intègre à un système de gestion de version (comme [git](#)), à un environnement de test et à d'autres éléments « au niveau du projet ».
- Si vous n'avez pas encore sélectionné d'IDE, envisagez les options suivantes :
 - [Visual Studio Code](#) (multiplateforme, gratuit).
 - [WebStorm](#) (multiplateforme, payant).
- Pour Windows, il y a aussi « Visual Studio », à ne pas confondre avec « Visual Studio Code ». « Visual Studio » est un éditeur Windows payant et puissant, bien adapté à la plate-forme .NET. C'est aussi bon en JavaScript. Il existe également une version gratuite [de Visual Studio Community](#).
- De nombreux IDE sont payés, mais ont une période d'essai. Leur coût est généralement négligeable par rapport au salaire d'un développeur qualifié, alors choisissez simplement le meilleur pour vous.

Environnement de travail & outils

Éditeurs légers

Les « éditeurs légers » ne sont pas aussi puissants que les IDE, mais ils sont rapides, élégants et simples.

Ils sont principalement utilisés pour ouvrir et éditer un fichier instantanément.

La principale différence entre un « éditeur léger » et un « IDE » est qu'un IDE fonctionne au niveau du projet, il charge donc beaucoup plus de données au démarrage, analyse la structure du projet si nécessaire, etc. Un éditeur léger est beaucoup plus rapide si nous n'avons besoin que d'un seul fichier.

En pratique, les éditeurs légers peuvent avoir beaucoup de plugins, y compris des analyseurs de syntaxe au niveau du répertoire et des autocompléteurs, de sorte qu'il n'y a pas de frontière stricte entre un éditeur léger et un IDE.

Les options suivantes méritent votre attention :

- [Atom](#) (multiplateforme, gratuit).
- [Sublime Text](#) (multiplateforme, shareware).
- [Notepad++](#) (Windows, gratuit).
- [Vim](#) et [Emacs](#) sont également cool si vous savez comment les utiliser.

Console développeur

21

Le code est sujet aux erreurs. Vous ferez très probablement des erreurs... Oh, de quoi je parle? Vous allez *absolument* faire des erreurs, du moins si vous êtes un humain, pas un robot.

Mais dans le navigateur, les utilisateurs ne voient pas les erreurs par défaut. Donc, si quelque chose ne va pas dans le script, nous ne verrons pas ce qui est cassé et ne pourrons pas le réparer.

Pour voir les erreurs et obtenir beaucoup d'autres informations utiles sur les scripts, des « outils de développement » ont été intégrés dans les navigateurs.

La plupart des développeurs se tournent vers Chrome ou Firefox pour le développement parce que ces navigateurs ont les meilleurs outils de développement. D'autres navigateurs fournissent également des outils de développement, parfois avec des fonctionnalités spéciales, mais jouent généralement à « rattraper » Chrome ou Firefox. Ainsi, la plupart des développeurs ont un navigateur « favori » et passent à d'autres si un problème est spécifique au navigateur.

Les outils de développement sont puissants; ils ont de nombreuses fonctionnalités. Pour commencer, nous allons apprendre à les ouvrir, à examiner les erreurs et à exécuter des commandes JavaScript.

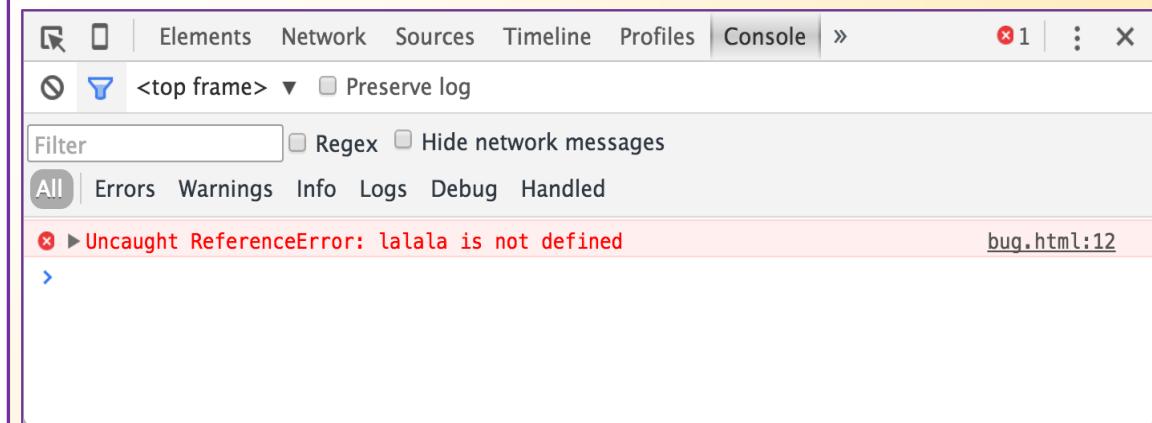
Google Chrome

Ouvrez le bogue de page.html.

Il y a une erreur dans le code JavaScript dessus. Il est caché aux yeux d'un visiteur régulier, alors ouvrons les outils de développement pour le voir.

Presser F12 ou Les outils de développement s'ouvrent par défaut sous l'onglet Console.

Cela ressemble un peu à ceci:



EP01:

22

- ▶ 1. Quelles sont les deux façons de s'assurer que votre code est compatible avec la plupart des navigateurs ?
- ▶ 2. Quelle est la différence entre ECMAScript et JavaScript ?
- ▶ 3. Citer quelques moteurs de navigateur de JavaScript?

Solutions EP01:

- ▶ 1. Pour vous assurer que votre code est rétro compatible avec les anciens navigateurs, vous pouvez transpilez votre code à l'aide d'un transpileur tel que Babel. Si vous développez une application pour utilisateurs d'un appareil ou d'un navigateur spécifique, il est sage de vérifier si une fonctionnalité ES particulière est pris en charge en vérifiant en ligne sur <https://www.caniuse.com>.
- ▶ 2. ECMA Script est un langage de script à usage général. Alors que JavaScript est un élément majeur implémentation de la spécification ECMAScript

Partie 2 :

les principes fondamentaux de javascript

- Affichage**
- Date**
- Variables**
- Types des données**
- Conversion des types**
- opérateurs mathématiques/logiques**

- Branche conditionnelle IF**
- Boucles**
- Switch**
- Fonctions**
- Setimeout/setInterval**
- Gestion d'erreurs**

Alert

24

- Il affiche un message et attend que l'utilisateur appuie sur "OK".
- La mini-fenêtre avec le message s'appelle une *fenêtre modale*. Le mot "modal" signifie que le visiteur ne peut pas interagir avec le reste de la page, appuyer sur d'autres boutons, etc., tant qu'il n'a pas traité la fenêtre. Dans ce cas – jusqu'à ce qu'ils appuient sur "OK".

➤ **Dans la plupart des cas, une nouvelle ligne implique un point-virgule. Mais « dans la plupart des cas » ne veut pas dire « toujours » !**

Il y a des cas où une nouvelle ligne ne signifie pas un point-virgule. Par exemple:

➤ Il est recommandé de placer des points-virgules entre les instructions, même si elles sont séparées par de nouvelles lignes. Cette règle est largement adoptée par la communauté. Notons encore une fois – *il est possible* de laisser de côté les points-virgules la plupart du temps. Mais il est plus sûr – surtout pour un débutant – de les utiliser.

1	alert(3 +
2	1
3	+ 2);

Balise Script :Hello World

25

La balise `<script>` contient du code JavaScript qui est automatiquement exécuté lorsque le navigateur traite la balise.

La balise `<script>` possède quelques attributs qui sont rarement utilisés de nos jours, mais que l'on peut encore trouver dans l'ancien code :

L'attribut type : `<script type=...>`

L'ancienne norme HTML, HTML4, exigeait un script pour avoir un type. Habituellement, c'était `type="text/javascript"`. Ce n'est plus nécessaire. En outre, la norme HTML moderne a totalement changé la signification de cet attribut. Maintenant, il peut être utilisé pour les modules JavaScript.

L'attribut language : `<script language=...>`

Cet attribut était destiné à montrer la langue du script. Cet attribut n'a plus de sens car JavaScript est le langage par défaut. Il n'est pas nécessaire de l'utiliser.

Commentaires avant et après les scripts.

Dans les livres et guides très anciens, vous pouvez trouver des commentaires à l'intérieur des balises `<script>`, comme ceci:

```
<script type="text/javascript"><!-- ... //--></script>
```

Cette astuce n'est pas utilisée dans javascript moderne. Ces commentaires cachent le code JavaScript des anciens

```
<!DOCTYPE HTML>
<html>
  <body>
    <p>Avant le script...</p>
    <script>
      alert( 'Hello, world!' );
    </script>
    <p>...apres le script.</p>
  </body>
</html>
```

Scripts externes:

mohamed@goumih.com

26

Si nous avons beaucoup de code JavaScript, nous pouvons le mettre dans un fichier séparé.
Les fichiers de script sont joints au code HTML avec `src` :

```
<script src="/path/to/script.js"></script>
```

Ici, `/path/to/script.js` est un chemin absolu vers le script à partir de la racine du site.

On peut également fournir un chemin relatif à partir de la page actuelle. Par exemple, `src="script.js"`, tout comme `src=". ./script.js"`, signifierait un fichier "`script.js`" dans le dossier actif

Nous pouvons également donner une URL complète. Par exemple:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js"></script>
```

Pour joindre plusieurs scripts, utilisez plusieurs balises :

```
<script src="/js/script1.js"></script> <script src="/js/script2.js"></script> ...
```

Si `src` est défini, le contenu du script est ignoré.

Une seule balise `<script>` ne peut pas contenir à la fois `src` et le code.

Cela ne fonctionnera pas :`<script src="file.js"> alert(1); // </script>`

Nous devons choisir soit un `<script src="...">` ou un `<script>` normal avec du code. L'exemple ci-dessus peut être divisé en deux scripts pour fonctionner :`<script src="file.js"></script>`
`<script> alert(1); </script>`

Affichage de texte JS

27

- ▶ JavaScript peut "afficher" les données de différentes manières :
- ▶ Écrire dans un élément HTML, en utilisant innerHTML.
- ▶ Écrire dans la sortie HTML à l'aide de document.write().
- ▶ Écrire dans une boîte d'alerte, en utilisant window.alert().
- ▶ Écrire dans la console du navigateur, en utilisant console.log().

```
<script>
document.getElementById("somme").innerHTML = 5 + 6;
document.write("la somme est :")
document.write(5+5);
console.log(5 + 6);
</script>
```

Strict mode :use strict

28

Pour activer pleinement toutes les fonctionnalités de JavaScript moderne, nous devrions commencer les scripts avec "use strict".

```
'use strict';
```

...

La directive doit être au sommet d'un script ou au début d'un corps de fonction.

Sans "use strict", tout fonctionne toujours, mais certaines fonctionnalités se comportent à l'ancienne, de manière "compatible".

Nous préférons généralement le comportement moderne.

Certaines fonctionnalités modernes du langage (telles que les classes que nous étudierons dans le futur) activent implicitement le mode strict.

Prompt /Confirm

29

prompt

La fonction **prompt** accepte deux arguments :

Elle affiche une fenêtre modale avec un message texte, un champ de saisie pour le visiteur et les boutons OK/Annuler.

title

Le texte à afficher au visiteur.

default

Un deuxième paramètre facultatif, la valeur initiale du champ d'entrée.

Le second paramètre est facultatif, mais si nous ne le fournissons pas

```
result = prompt(title, [default]);
```

```
let age = prompt('How old are you?', 100);
alert(`You are ${age} years old!`);
```

confirm

La fonction **confirm** affiche une fenêtre modale avec une question et deux boutons : OK et Annuler.

Le résultat est **true** si vous appuyez sur OK et **false** dans le cas contraire:

```
let stg = confirm("vous etes un stagiaire?");
alert( stg ); // true si OK est pressé
```

Commentaires

30

Les commentaires d'une ligne commencent par deux barres obliques `//`.

Le reste de la ligne est un commentaire. Il peut occuper une ligne complète ou suivre une déclaration.

Les commentaires multilignes commencent par une barre oblique et un astérisque `/*` et se terminent par un astérisque et une barre oblique `*/`.

Utilisez des raccourcis clavier!

Dans la plupart des éditeurs, une ligne de code peut être commentée en appuyant sur la touche `Ctrl+ /` raccourci clavier pour un commentaire d'une seule ligne et quelque chose comme `alt+Shift+A` ou `Ctrl+k ctrl+c` pour les commentaires multilignes (sélectionnez un morceau de code et appuyez sur le raccourci clavier).

```
<script src="alert.js">
// commentaire
alert('World'); //commentaires
/*commentaires
dans plusieurs
lignes*/
```

Date et Temps

31

Pour créer un nouvel objet Date, appelez new Date () avec l'un des arguments suivants:

```
let now = new Date();
alert( now ); // affiche la date/heure actuelle
```

Crée un objet Date avec l'heure correspondant au nombre de millisecondes (1/1000 de seconde) écoulée après le 1er janvier 1970 UTC.

Il existe de nombreuses méthodes pour accéder à l'année, au mois, etc. à partir de l'objet Date.

getFullYear():Obtenir l'année (4 chiffres)

getMonth():Obtenir le mois, **de 0 à 11**.

getDate():Obtenir le jour du mois, de 1 à 31.

getHours(), getMinutes(), getSeconds(), getMilliseconds()

,getDay()

Obtenir l'heures / les minutes / les secondes / les millisecondes

// 0 signifie 01.01.1970 UTC+0

```
let Jan01_1970 = new Date(0);
alert( Jan01_1970 );
```

// maintenant, ajoutez 24 heures, cela devient 02.01.1970 UTC+0

```
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 );
```

`new Date(2011, 0, 1, 0, 0, 0, 0);` // // 1 Jan 2011, 00:00:00

`new Date(2011, 0, 1);` // la même chose car les heures etc sont égales à 0 par défaut

getTime()Renvoie l'horodatage de la date – nombre de millisecondes écoulées à partir du 1er janvier 1970 UTC + 0.

getTimezoneOffset()Renvoie la différence entre le fuseau horaire local et l'heure UTC, en minutes :

```
// date actuel
let date = new Date();
```

```
// l'heure dans votre fuseau horaire
alert( date.getHours() );
```

```
// l'heure dans le fuseau horaire UTC
alert( date.getUTCHours() );
```

Date et Temps

32

```
let date = new Date();
alert(+date); // le nombre de millisecondes, identique à date.getTime ()
```

La méthode `Date.parse(str)` peut lire une date provenant d'une chaîne de caractères

Le format de la chaîne de caractères doit être: YYYY-MM-DDTHH:mm:ss.sssZ, où:

- YYYY-MM-DD – est la date : année-mois-jour.
- Le caractère "T" est utilisé comme délimiteur.
- HH:mm:ss.sss – correspond à l'heure : heures, minutes, secondes et millisecondes.
- La partie optionnelle Z indique le fuseau horaire au format +-hh:mm. Une seule lettre Z qui signifierait UTC + 0.

Des variantes plus courtes sont également possibles, telles que AAAA-MM-JJ ou AAAA-MM ou même AAAA.

L'appel à `Date.parse(str)` analyse la chaîne au format indiqué et renvoie l'horodatage (nombre de millisecondes à compter du 1er janvier 1970 UTC + 0).

Si le format n'est pas valide, renvoie NaN.

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');

alert(ms); // 132761110417 (horodatage)

let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );

alert(date);
```

EP02:

33

- ▶ Créez une page qui affiche un message Je suis JavaScript !
 - » avec **une alerte de script externe**
- ▶ **Créer une alerte « Hello world » divisé en deux avec point-virgule et sans !**
- ▶ Affichez la date du jour dans la console du navigateur
Créez une Date pour la date: 28 février 2023, 8h30. Le fuseau horaire est local.
Montrez-le en utilisant alert
- ▶ **Afficher votre nom avec document.innerHTML**

Les variables

34

Une variable est un “stockage nommé” pour les données. Nous pouvons utiliser des variables pour stocker des goodies, des visiteurs et d’autres données.

Pour créer une variable en JavaScript, nous devons utiliser le mot-clé `let`.

L'instruction ci-dessous crée (autrement dit: *declare*) une variable avec le nom “message” :

```
let message ;  
message="DD";  
alert (message);
```

Nous pouvons également déclarer plusieurs variables sur une seule ligne :

```
1 let user = 'John', age = 25, message = 'Hello';
```

Cela peut sembler plus court, mais ce n'est pas recommandé.

Pour une meilleure lisibilité, veuillez utiliser une seule ligne par variable.

La variante multiligne est un peu plus longue, mais plus facile à lire :

```
1 let user = 'John';  
2 let age = 25;  
3 let message = 'Hello';
```

```
1 let user = 'John'  
2 , age = 25  
3 , message = 'Hello';
```

Dans les anciens scripts, vous pouvez également trouver un autre mot-clé : `var` au lieu de `let` :

```
var message = 'Hello';
```

Le mot-clé `var` est *presque* identique à `let`. Il déclare également une variable, mais d'une manière légèrement différente, à la mode “old school”.

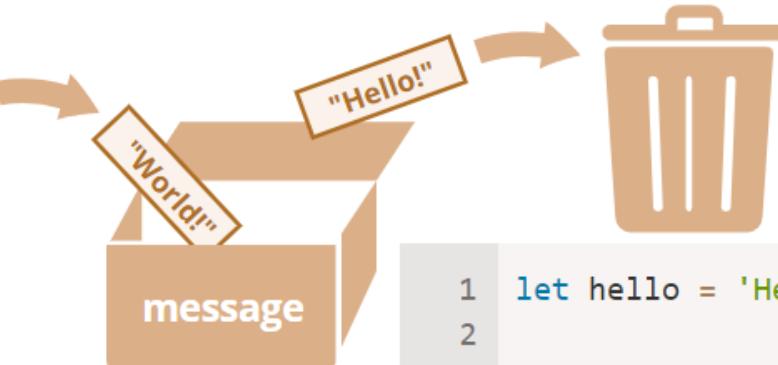
Variable :Une analogie

mohamed@goumih.com

35



```
1 let message;  
2  
3 message = 'Hello!';  
4  
5 message = 'World!'; // valeur changée  
6  
7 alert(message);
```



```
1 let hello = 'Hello world!';  
2  
3 let message;  
4  
5 // copier 'Hello world' de hello vers message  
6 message = hello;  
7  
8 // maintenant les deux variables contiennent les mêmes données  
9 alert(hello); // Hello world!  
10 alert(message); // Hello world!
```

Une variable ne doit être déclarée qu'une seule fois.

Une déclaration répétée de la même variable est une erreur :

Variables

Il existe deux limitations pour un nom de variable en JavaScript :

1. Le nom ne doit contenir que des lettres, des chiffres, des symboles \$ et _.
2. Le premier caractère ne doit pas être un chiffre:

```
Let Nom ;  
Let a1,b32;  
Let $x;  
Let _def;  
alert($+_)
```

La casse est importante

Des variables nommées pomme and PomMe – sont deux variables différentes.

Les lettres non latines sont autorisées mais non recommandées

Il est possible d'utiliser n'importe quel langage,
y compris les lettres cyrilliques ou même les hiéroglyphes, comme ceci :

```
let имя = '...';  
let 我 = '...';  
Let اسم='mohamed'
```

Noms réservés

Il existe une liste de mots réservés, qui ne peuvent pas être utilisés comme noms de variables, car ils sont utilisés par le langage lui-même.
Par exemple, les mots `let`, `class`, `return`, `function` sont réservés.

Constantes

37

Pour déclarer une constante (non changeante), on peut utiliser **Const** plutôt que **let** :

```
const dateNaissance = '18.04.2000';
dateNaissance = '18.04.1990';
// erreur, ne peut pas réaffecter la constante !
```

Lorsqu'un programmeur est certain que la variable ne doit jamais changer, il peut utiliser **const** pour le garantir et également le montrer clairement à tout le monde.

Il existe une pratique répandue d'utiliser des constantes comme alias pour des valeurs difficiles à mémoriser, qui sont connues avant leur exécution. Ces constantes sont nommées en utilisant des majuscules et des underscores.

Par exemple, créons des constantes pour les couleurs au format dit "web" (hexadécimal) :

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ... quand il faut choisir une couleur
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

A retenir:

Nous pouvons déclarer des variables pour stocker des données. Cela peut être fait en utilisant **var** ou **let** ou **const**.

- **let** – est une déclaration de variable moderne.
- **var** – est une déclaration de variable old-school. Normalement, nous ne l'utilisons pas du tout juste au cas où vous en auriez besoin.
- **const** – est équivalent à **let**, mais la valeur de la variable ne peut pas être modifiée. Les variables doivent être nommées d'une manière qui nous permet de comprendre facilement ce qui est à l'intérieur.

EP03 :

39

A

- 1.Déclarez 3 variables : nom ,prenom et nomComplet.
- 2.Assignez votre nom ,prenom aux variables nom ,prenom
- 3.Copiez la valeur de nom +prenom dans nomComplet.
- 4.Afficher la valeur de nomComplet en utilisant :alert et innerHtml

B

Créez la variable avec le nom de notre planète. Comment nommeriez-vous une telle variable ?

C

Déclarer variables a,b,c avec des valeurs ,et par la suite mettre la valeur De a dans b et a dans b à l'aide du variable c ;

Les types de données

- Une valeur en JavaScript est toujours d'un certain type. Par exemple, une chaîne de caractères ou un nombre.
- Il existe 8 types de données de base en JavaScript. Ici, nous les couvrirons en général et dans les prochains chapitres, nous parlerons de chacun d'eux en détail.
- Nous pouvons mettre n'importe quel type dans une variable. Par exemple, une variable peut à un moment être une chaîne de caractères puis stocker un nombre :

Les langages de programmation qui permettent de telles choses sont appelés “**typés dynamiquement**”, ce qui signifie qu'il existe des types de données, mais que les variables ne sont liées à aucun d'entre eux.

```
1 // pas d'erreur
2 let message = "hello";
3 message = 123456;
```

```
1 let n = 123;
2 n = 12.345;
```

Le type **number** sert à la fois à des nombres entiers et à des nombres à virgule flottante. Il existe de nombreuses opérations pour les nombres, par ex. multiplication *, division /, addition +, soustraction - et ainsi de suite.

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

- **Infinity** représente 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
```

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

Number

Nombres

41

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  
  
alert( "pas un nombre" / 2 + 5 ); // NaN
```

BigInt

En JavaScript moderne, il existe deux types de nombres :

En JavaScript, le type “number” ne peut pas représenter des valeurs entières supérieures à $(2^{53}-1)$ (c'est 9007199254740991), ou moins que $-(2^{53}-1)$ pour les chiffres négatifs. C'est une limitation technique causée par leur représentation interne.

Dans la plupart des cas, cela suffit, mais parfois nous avons besoin de très gros nombres, par exemple pour la cryptographie ou les horodatages à la microseconde.

BigInt a récemment été ajouté au langage pour représenter des entiers de longueur arbitraire.

Une valeur BigInt est créé en ajoutant n à la fin d'un entier :

```
// le "n" à la fin signifie que c'est un BigInt  
const bigInt = 1234567890123456789012345678901234567890n;
```

À l'heure actuelle, BigInt est pris en charge dans Firefox/Chrome/Edge/Safari, mais pas dans IE.

String

42

En JavaScript, type caractère n'existe pas. Il n'y a qu'un seul type: **string**. Une chaîne de caractères peut être composée de zéro caractère (être vide), d'un caractère ou de plusieurs d'entre eux.

► Une chaîne de caractères en JavaScript doit être entre guillemets.

En JavaScript, il existe 3 types de guillemets.

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

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

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 :

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 !

```
let name = "John";

// une variable encapsulée
alert(`Hello, ${name}!`); // Hello, John!

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

Boolean (type logique)

43

Le type booléen n'a que deux valeurs: **true** et **false**.

Ce type est couramment utilisé pour stocker des valeurs oui / non: **true** signifie "oui, correct" et **false** signifie "non, incorrect".

Par exemple :

```
let valider=true;  
let rattrapage=false;  
let a=4>1;  
alert(a)
```

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

```
let demarrer=null;
```

La valeur spéciale **undefined** se distingue des autres. C'est un type à part entière, comme null. La signification de **undefined** est "la valeur n'est pas attribuée". Si une variable est déclarée mais non affectée, alors sa valeur est exactement **undefined** :

```
let age;  
  
alert(age); // affiche "undefined"
```

```
let age = 100;  
  
// change the value to undefined  
age = undefined;  
  
alert(age); // "undefined"
```

Object - Symbol -typeof

44

Le type **object** est spécial:

Tous les autres types sont appelés “primitifs”, car leurs valeurs ne peuvent contenir qu’une seule chose (que ce soit une chaîne de caractères, un nombre ou autre). À contrario, les objets servent à stocker des collections de données et des entités plus complexes.

Étant aussi important, les objets méritent un traitement spécial.

Le type **symbol** est utilisé pour créer des identificateurs uniques pour les objets. Nous devons le mentionner ici par souci d’exhaustivité, mais nous allons le voir en détails après avoir étudié les objets

L’opérateur **typeof** renvoie le type de l’argument. Il est utile lorsqu’on souhaite traiter différemment les valeurs de différents types ou de faire une vérification rapide.

Il supporte deux formes de syntaxe :

- 1.Sous forme d’opérateur : `typeof x`.
- 2.En style de fonction : `typeof(x)`.

En d’autres termes, cela fonctionne à la fois avec ou sans parenthèses. Le résultat est le même.

L’appel `typeof x` renvoie une chaîne de caractères avec le nom du type :

```
typeof undefined // "undefined"
typeof 0 // "number"
typeof 10n // "bigint"
typeof true // "boolean"
typeof "foo" // "string"
typeof Symbol("id") // "symbol"
typeof Math // "object" (1)
typeof null // "object" (2)
typeof alert // "function" (3)
```

A RETENIR :

45

Il existe 8 types de données de base en JavaScript.

- **number** pour les nombres de toute nature : entier ou virgule flottante, les nombres entiers sont limités à $\pm(2^{53}-1)$.
- **bigint** pour des nombres entiers de longueur arbitraire.
- **string** pour les chaînes de caractères. Une chaîne de caractères peut avoir zéro ou plusieurs caractères, il n'y a pas de type à caractère unique distinct.
- **boolean** pour true/false (vrai/faux).
- **null** pour les valeurs inconnues – un type autonome qui a une seule valeur null.
- **undefined** pour les valeurs non attribuées – un type autonome avec une valeur unique undefined.
- **object** pour des structures de données plus complexes.
- **symbol** pour les identifiants uniques.

L'opérateur **typeof** nous permet de voir quel type est stocké dans la variable.

- Deux formes : **typeof x** ou **typeof(x)**.
- Renvoie une chaîne de caractères avec le nom du type, comme "string".
- Pour **null** il renvoie "object" – C'est une erreur dans le langage, ce n'est pas un objet en fait.

Les conversions de types

46

La conversion **String** se produit lorsque nous avons besoin de la forme chaîne de caractères d'une valeur.

Par exemple, **alert(value)** le fait pour afficher la valeur.

Nous pouvons également utiliser un appel de fonction **String(value)** pour ça :

```
let value = true;
alert(typeof value); // boolean

value = String(value); // maintenant la valeur est une chaîne de caractères "true"
alert(typeof value); // string
```

La conversion numérique se produit automatiquement dans les fonctions et les expressions mathématiques.

Par exemple, lorsque la division / est appliquée à des non-numéros :

```
alert( "6" / "2" ); // 3, les chaînes de caractères sont converties en
```

Nous pouvons utiliser une fonction **Number(value)** pour convertir explicitement une valeur :

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

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

Les conversions de types

47

Une conversion explicite est généralement requise lorsque nous lisons une valeur à partir d'une source basée sur des chaînes de caractères, par exemple un champ texte, mais qu'un nombre doit être entré. Si la chaîne de caractères n'est pas un nombre valide, le résultat de cette conversion est NaN :

```
let age = Number("une chaîne de caractères arbitraire au lieu d'un nombre")
alert(age); // NaN, la conversion a échoué
```

Valeur	Devient ...	Règles de conversion numériques :
undefined	Nan	
null	0	
true et false	1 et 0	
string		Les espaces blancs du début et de la fin sont supprimés. Ensuite, si la chaîne restante est vide, le résultat est 0. Sinon, le nombre est «lu» dans la chaîne. Une erreur donne NaN .

```
1 alert( Number(" 123  ") ); // 123
2 alert( Number("123z") ); // NaN (erreur de lecture d'un nombre à "z")
3 alert( Number(true) ); // 1
4 alert( Number(false) ); // 0
```

Les valeurs qui sont intuitivement “vides”, comme 0, une chaîne de caractères, null, undefined NaN deviennent false. Les autres valeurs deviennent true. La conversion booléenne est la plus simple.

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

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

Opérateurs mathématiques

48

De nombreux opérateurs nous sont connus de l'école. Ce sont les additions +, multiplications *, soustractions - et ainsi de suite.

Termes: "unaire", "binaire", "opérande"

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

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

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

Reste % (Modulo)

L'opérateur reste %, malgré son apparence, n'est pas lié aux pourcentages.

Le résultat de $a \% b$ est le reste de la division entière de a par b .

Par exemple:

```
alert( 5 % 2 ); // 1, le reste de 5 divisé par 2
alert( 8 % 3 ); // 2, le reste de 8 divisé par 3
```

Opérateurs mathématiques

49

- ▶ **Exponentiation **** : L'opérateur d'exponentiation $a^{**} b$ multiplie a par lui-même b fois. En mathématiques à l'école, nous écrivons cela a^b .

```
alert( 2 ** 2 ); // 22 = 4
alert( 2 ** 3 ); // 23 = 8
alert( 2 ** 4 ); // 24 = 16
```

```
alert( 4 ** (1/2) ); // 2 (la puissance de 1/2 équivaut à une racine carré)
alert( 8 ** (1/3) ); // 2 (la puissance de 1/3 équivaut à une racine cubiq
```

Concaténation de chaînes de caractères, binaire +

Habituellement, l'opérateur + additionne des chiffres.

Mais si le binaire + est appliqué aux chaînes de caractères, il les fusionne (concatène) :

```
let s = "my" + "string";
alert(s); // mystring
```

```
alert(2 + 2 + '1' );
```

```
alert('1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Le plus + existe sous deux formes. La forme binaire que nous avons utilisée ci-dessus et la forme unaire. L'inaire plus ou, en d'autres termes, l'opérateur plus + appliqué à une seule valeur, ne fait rien avec les nombres, mais si l'opérande n'est pas un nombre, alors il est convertit en nombre.

```
let x = 1;
alert( +x ); // 1
```

```
alert( 6 - '2' );
alert( '6' / '2' )
```

EP04:

mohamed@goumih.com

50

Concaténer ses 2 valeurs :

► **let apples = "2"; let oranges = "3";**

► **Modifier le code pour afficher la somme de 2+3**

```
let apples = "2";
let oranges = "3";

// les deux valeurs converties en nombres avant le binaire plus
alert( +apples + +oranges ); // 5

// c'est équivalent à cette variante plus longue
// alert( Number(apples) + Number(oranges) ); // 5
```

a = b = c = 2 + 2;

► Quels sont les Valeurs de
a,b,c,n & C?

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

```
let a = 1;
let b = 2;
let c = 3 - (a = b + 1);
```

Incrémantion / décrémentation

51

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 :

- **Incrémantion ++** augmente une variable de 1 . **Décrémentation --** diminue une variable de 1 :

```
let counter = 2;    let counter = 2;  
counter++;        // counter--;      //  
alert( counter ); alert( counter );
```

forme préfixe

```
let counter = 1;  
let a = ++counter;
```

forme postfixe

```
let counter = 1;  
let a = counter++;
```

```
let counter = 1;  
alert( 2 * ++counter );
```

```
let counter = 1;  
alert( 2 * counter++ );
```

EP05:

52

Quelles sont les valeurs finales de toutes les variables a, b, c et d après le code ci-dessous ?

```
let a = 1, b = 1;  
  
let c = ++a; // ?  
let d = b++; // ?
```

```
let a = 2;  
  
let x = 1 + (a *= 2);
```

Quels sont les résultats de ces expressions ?

```
"" + 1 + 0  
"" - 1 + 0  
true + false  
6 / "3"  
"2" * "3"  
4 + 5 + "px"  
"$" + 4 + 5  
"4" - 2  
"4px" - 2  
" -9 " + 5  
" -9 " - 5  
null + 1  
undefined + 1  
" \t \n" - 2
```

Voici un code qui demande à l'utilisateur deux nombres et affiche leur somme. Cela ne fonctionne pas correctement. La sortie dans l'exemple ci-dessous est 12 (pour les valeurs d'invite par défaut). Pourquoi ? Réparez-le. Le résultat doit être 3.

```
let a = prompt("First number?", 1);  
let b = prompt("Second number?", 2);  
  
alert(a + b); // 12
```

Comparaisons

53

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 : $a \geq b$, $a \leq b$.
- 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 \neq , mais en JavaScript elle est écrite comme une assignation avec un signe d'exclamation : $a != b$.

Un opérateur d'égalité stricte === vérifie l'égalité sans conversion de type.

En d'autres termes, si a et b sont de types différents, alors $a === b$ renvoie immédiatement `false` sans tenter de les convertir.

Essayons :

```
alert( 0 === false ); // false, parce que les types sont différents
```

```
alert( 2 > 1 ); // true (correct)
alert( 2 == 1 ); // false (faux)
alert( 2 != 1 ); // true (correct)
```

```
let result = 5 > 4; // attribue le résultat de la comparaison
alert( result ); // true
```

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

```
alert( '2' > 1 ); // true, la chaîne '2' devient un numéro 2
alert( '01' == 1 ); // true, chaîne '01' devient un numéro 1
alert( true == 1 ); // true
alert( false == 0 ); // true
```

EP06:

54

Quel sera le résultat pour les expressions suivantes :

5 > 4

"apple" > "pineapple"

"2" > "12"

undefined == null

undefined === null

null == "\n0\n"

null === +"\\n0\\n"

Branche conditionnelle : if, '?'

55

L'instruction **if(...)** évalue une condition entre parenthèses et, si le résultat est **true**, exécute un bloc de code.

```
if (year == 2015) {
    alert( "That's correct!" );
    alert( "You're so smart!" );
}
```

L'instruction **if** peut contenir un bloc optionnel “**else**”. Il s'exécute lorsque la condition est fausse. Parfois, nous aimerais tester plusieurs variantes d'une condition. Il y a une clause **else if** pour cela .

```
if (year == 2015) {
    alert( 'You guessed it right!' );
} else {
    alert( 'How can you be so wrong?' ); // toute autre valeur que 2015
}
```

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

L'opérateur “**ternaire**” ou “**point d'interrogation**” nous permet de le faire plus rapidement et plus simplement.

L'opérateur est représenté par un point d'interrogation **?**. Appelé aussi “**ternaire**” parce que l'opérateur a trois opérandes. C'est en fait le seul et unique opérateur en JavaScript qui en a autant.

La syntaxe est : **let result = condition ? value1 : value2**

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

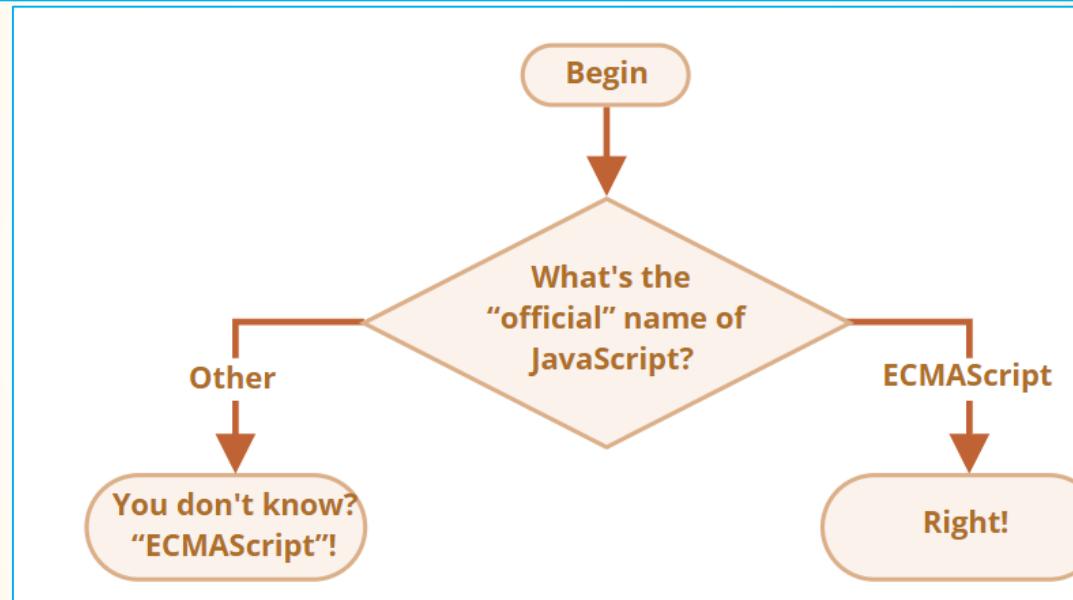
let message = (age < 3) ? 'Hi, I'm a baby!' :
               (age < 18) ? 'Hello!' :
               (age < 100) ? 'Greetings!' :
               'What an unusual age!';

alert( message );
```

EP07:

56

En utilisant la construction `if..else`, écrivez le code qui demande : ‘Quel est le nom “officiel” de JavaScript?’ Si le visiteur entre “ECMAScript”, alors éditez une sortie “Bonne réponse !”, Sinon – retourne “Ne sait pas ? ECMAScript!”



En utilisant `if..else`, écrivez le code qui obtient un numéro via le `prompt`, puis l'affiche en `alert` :

- 1, si la valeur est supérieure à zéro,
- -1, si inférieur à zéro,
- 0, si est égal à zéro.

Dans cet exercice, nous supposons que l'entrée est toujours un nombre.

EP08:

57

Réécrire ce `if..else` en utilisant plusieurs opérateurs ternaires '`?`'.

```
let message;

if (login == 'Employee') {
    message = 'Hello';
} else if (login == 'Director') {
    message = 'Greetings';
} else if (login == '') {
    message = 'No login';
} else {
    message = '';
}
```

Opérateurs logiques

58

Il y a trois opérateurs logiques en JavaScript : **|| (OR)**, **&& (AND)**, **! (NOT)**, **?? (Coalescence des nulles)**.

Nous couvrons ici les trois premiers, l'opérateur **??** Par la suite.

Bien qu'ils soient appelés "logiques", ils peuvent être appliqués à des valeurs de tout type, pas seulement booléennes.

Le résultat peut également être de tout type.

||: (OR)

L'opérateur "OR" est représenté avec deux symboles de ligne verticale :

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

```
result = a || b;
```

```
result = value1 || value2 || value3;
```

La plupart du temps, OR **||** est utilisé dans une instruction **if** pour tester si l'une des conditions données est correcte.

```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert( 'The office is closed.' );
}
```

```
if (1 || 0) { // fonctionne comme si ( true || false )
  alert( 'truthy!' );
}
```

Opérateurs logiques :&& ,!

59

L'opérateur AND est représenté avec deux esperluettes **&&** :

```
result = a && b;
```

```
result = value1 && value2 && value3;
```

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

AND retourne **true** si les deux opérandes sont **true** et **false** dans les autres cas.

La précédence de AND && est supérieure à OR ||

La priorité de l'opérateur AND **&&** est supérieure à OR **||**.

Donc, le code **a && b || c && d** est essentiellement le même que si **&&** était entre parenthèses: **(a && b) || (c && d)**.

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert( 'Time is 12:30' );
}
```

!(NOT):

L'opérateur booléen NOT est représenté par un point d'exclamation **!**.

L'opérateur accepte un seul argument et effectue les opérations suivantes :

- 1.Convertit l'opérande en type booléen : **true/false**.
- 2.Renvoie la valeur inverse.

Par exemple :

```
result = !value;
```

Un double NOT **!!** est parfois utilisé pour convertir une valeur en type booléen :

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

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

EP09:

60

Ecrivez une condition "if" pour vérifier que l'age est compris entre 14 et 90 ans inclus.
"Inclus" signifie que l'age peut atteindre les 14 ou 90 ans.

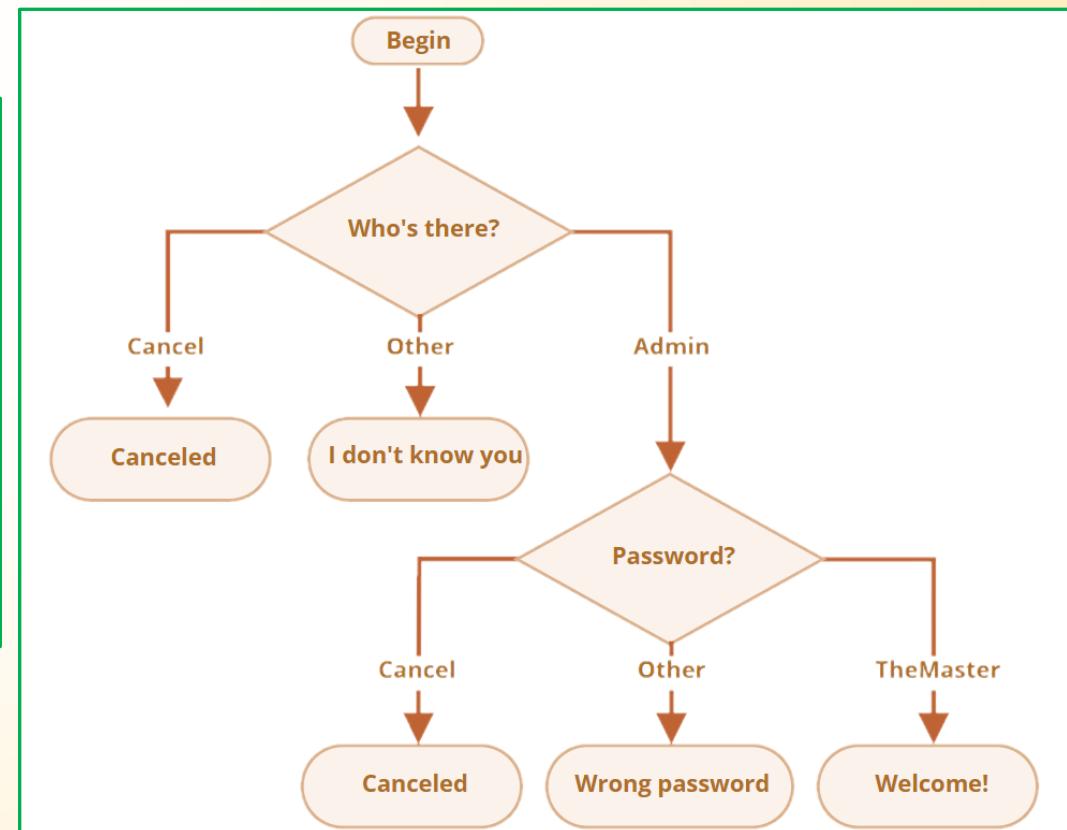
Ecrivez une condition if pour vérifier que l'age n'est PAS compris entre 14 et 90 ans inclus.
Créez deux variantes: la première utilisant NOT !, La seconde – sans ce dernier.

Écrivez le code qui demande une connexion avec prompt .

Si le visiteur entre "Admin" , puis prompt pour un mot de passe, si l'entrée est une ligne vide ou Esc – affichez "Canceled", s'il s'agit d'une autre chaîne de caractères – alors affichez "I don't know you". Le mot de passe est vérifié comme suit :

- S'il est égal à "TheMaster", alors affichez "Welcome!" ,
- Une autre chaîne de caractères – affichez "Wrong password",
- Pour une chaîne de caractères vide ou une entrée annulée, affichez "Canceled".

Veuillez utiliser des blocs IF imbriqués



L'opérateur de coalescence des nuls '??'

mohamed@goumih.com

61

L'opérateur de coalescence des nuls est écrit sous la forme de deux points d'interrogation ??.

Comme il traite null et undefined de la même manière.

Nous dirons qu'une expression est "définie" lorsqu'elle n'est ni null ni undefined.

Le résultat de a ?? b est :

- si a est défini, alors a,
- si a n'est pas défini, alors b.

```
let user;
```

```
alert(user ?? "Anonymous"); // Anonymous (user not defined)
```

```
let user = "John";
```

```
alert(user ?? "Anonymous"); // John (user defined)
```

```
let firstName = null;  
let lastName = null;  
let nickName = "Supercoder";  
  
// affiche la première valeur définie:  
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder  
undefined
```

- L'expression height || 100 vérifie que height est une valeur fausse, et c'est 0, elle est fausse en effet.
 - donc le résultat de || est le deuxième argument, 100.

```
// configurer height=100, si height est null ou undefined  
height = height ?? 100;
```

```
let height = 0; alert(height || 100); // 100  
alert(height ?? 100); // 0
```

Boucles : while et Do While

mohamed@goumih.com

62

► Les boucles permettent de répéter plusieurs fois la même partie du code.

La boucle "while"

La boucle while a la syntaxe suivante :

```
while (condition) {  
    // code  
    // appelé "loop body" ("corps de boucle")  
}
```

Une unique exécution du corps de la boucle est appelée **une itération**.

```
let i = 0;  
while (i < 3) { // affiche 0, puis 1, puis 2  
    alert( i );  
    i++;  
}
```

```
let i = 3;  
while (i) { // quand i devient 0, la condition devient fausse  
    alert( i );  
    i--;  
}
```

La boucle "do...while"

La vérification de la condition peut être déplacée *sous* le corps de la boucle en utilisant la syntaxe do..while :

```
do {  
    // corps de la boucle  
} while (condition);
```

```
let i = 0;  
do {  
    alert( i );  
    i++;  
} while (i < 3);
```

Boucle: For

```
let i = 0;
for (;;) {
  for (; i < 3;) { // répète sans limites
    alert( i++ );
  }
}
```

```
let i = 0; // nous avons i déjà déclaré et assigné
for (; i < 3; i++) { // pas besoin de "début"
  alert( i ); // 0, 1, 2
}
```

```
let i = 0;

for (i = 0; i < 3; i++) { // utiliser une variable existante
  alert(i); // 0, 1, 2
}
alert(i); // 3, visible, car déclaré en dehors de la boucle
```

```
for (début; condition; étape) {
  // ... corps de la boucle ...
}
```

```
for (let i = 0; i < 3; i++) { // affiche 0, puis 1, puis 2
  alert(i);
}
```

Briser la boucle

Normalement, la boucle sort quand la condition devient fausse.
Mais nous pouvons forcer la sortie à tout moment. Il y a une directive spéciale appelée Break

```
let sum = 0;

while (true) {
  let value = +prompt("Entrez un nombre", '');
  if (!value) break; // (*)
  sum += value;
}
alert( 'Sum: ' + sum );
```

Boucle :for

64

La directive **continue** est une “version plus légère” de **Break**

- ▶ Elle a n’arrête pas toute la boucle. Au lieu de cela, elle arrête l’itération en cours et force la boucle à en démarrer
- ▶ une nouvelle (si la condition le permet).
- ▶ Nous pouvons l’utiliser si nous avons terminé l’itération en cours et aimerais passer à la suivante.

```
for (let i = 0; i < 10; i++) {  
  
    // si vrai, saute le reste du corps  
    if (i % 2 == 0) continue;  
  
    alert(i); // 1, ensuite 3, 5, 7, 9  
}
```

EP10:

65

Utilisez la boucle **for** pour afficher les nombres pairs de 2 à 10.

Réécrivez le code en modifiant la boucle **for** en **while** sans modifier son comportement (la sortie doit rester la même).

Ecrivez une boucle qui demande un nombre supérieur à 100. Si le visiteur saisit un autre numéro, demandez-lui de le saisir à nouveau.

La boucle doit demander un numéro jusqu'à ce que le visiteur saisisse un nombre supérieur à 100 ou annule l'entrée/entre une ligne vide.

Ici, nous pouvons supposer que le visiteur ne saisit que des chiffres. Il n'est pas nécessaire de mettre en œuvre un traitement spécial pour une entrée non numérique dans cette tâche.

Un nombre entier supérieur à 1 est appelé un Nombre premier s'il ne peut être divisé sans reste par rien d'autre que 1 et lui-même.

En d'autres termes, $n > 1$ est un nombre premier s'il ne peut être divisé de manière égale par autre chose que 1 et n .

Par exemple, 5 est un nombre premier, car il ne peut pas être divisé sans reste par 2, 3 et 4.

Écrivez un code qui produit les nombres premiers dans l'intervall e 2 à n.

Pour $n = 10$, le résultat sera 2,3,5,7

switch

Une instruction switch peut remplacer plusieurs vérification if.
Cela donne un moyen plus descriptif de comparer une valeur avec plusieurs variantes.
Le switch a un ou plusieurs blocs case (cas) et une valeur par défaut facultative.

```
switch(x) {  
    case 'value1': // si (x === 'value1')  
        ...  
        [break]  
  
    case 'value2': // si (x === 'value2')  
        ...  
        [break]  
  
    default:  
        ...  
        [break]  
}
```

```
let a = 2 + 2;  
  
switch (a) {  
    case 4:  
        alert('Right!');  
        break;  
  
    case 3: // (*) grouped two cases  
    case 5:  
        alert('Wrong!');  
        alert("Why don't you take a math class?");  
        break;  
  
    default:  
        alert('The result is strange. Really.');  
}
```

```
let a = 2 + 2;  
  
switch (a) {  
    case 3:  
        alert( 'Too small' );  
        break;  
    case 4:  
        alert( 'Exactly!' );  
        break;  
    case 5:  
        alert( 'Too big' );  
        break;  
    default:  
        alert( "I don't know such values" );  
}
```

EP11:

67

Écrivez le code en utilisant `if..else` qui correspondrait au `switch` suivant :

```
let a = 2 + 2;

switch (a) {
  case 4:
    alert('Right!');
    break;

  case 3: // (*) grouped two cases
  case 5:
    alert('Wrong!');
    alert("Why don't you take a math class?");
    break;

  default:
    alert('The result is strange. Really.');
}
```

Réécrivez le code ci-dessous en utilisant une seule instruction `switch`:

```
let a = +prompt('a?', '')

if (a == 0) {
  alert( 0 );
}

if (a == 1) {
  alert( 1 );
}

if (a == 2 || a == 3) {
  alert( '2,3' );
}
```

Créez un programme JavaScript qui demande à l'utilisateur de saisir deux nombres à l'aide de la fonction **prompt**. Ensuite, demandez à l'utilisateur de choisir une opération dans la liste suivante : addition (+), soustraction (-), multiplication (*) ou division (/).

En fonction du choix d'opération de l'utilisateur, effectuez le calcul mathématique correspondant sur les deux nombres et affichez le résultat à l'utilisateur à l'aide de la fonction **alert**.

Après avoir affiché le résultat, demandez à l'utilisateur s'il souhaite effectuer un autre calcul.

Si l'utilisateur répond "oui", répétez le processus depuis le début. Si l'utilisateur répond "non", terminez le programme.

Assurez-vous d'utiliser des boucles et des instructions conditionnelles pour implémenter ce programme.

Créez un programme JavaScript qui génère une table de multiplication pour un nombre saisi par l'utilisateur. Le programme doit demander à l'utilisateur d'entrer un nombre compris entre 1 et 10 à l'aide de la fonction **prompt**. Si l'utilisateur entre un nombre en dehors de cette plage, le programme doit afficher un message d'erreur et demander à l'utilisateur d'entrer un nombre valide. Une fois que l'utilisateur a entré un nombre valide, le programme doit générer une table de multiplication pour ce nombre, montrant les produits de ce nombre avec tous les nombres entiers entre 1 et 10. La table doit être affichée à l'aide de la fonction **alert** ou **console.log**.

Après avoir affiché la table, demandez à l'utilisateur s'il souhaite générer une autre table de multiplication. Si l'utilisateur répond "oui", répétez le processus depuis le début. Si l'utilisateur répond "non", terminez le programme.

Fonctions

69

Les fonctions sont les principales “composantes” du programme. Ils permettent au code d’être appelé plusieurs fois sans répétition.

Nous avons déjà vu des exemples de fonctions intégrées, telles que `alert(message)`, `prompt(message, default)` et `confirm(question)`.

Mais nous pouvons aussi créer nos propres fonctions.

Pour créer une fonction, nous pouvons utiliser une *déclaration de fonction*.

Cela ressemble à ceci :

```
function showMessage() {
    alert( 'Hello everyone!' );
}
```

Le mot-clé `function` commence en premier, puis le *nom de la fonction*, puis une liste de *paramètres* entre les parenthèses (séparés par des virgules, vides dans l'exemple ci-dessus) et enfin le code de la fonction, également appelé “le *corps de la fonction*”, entre des accolades.

```
function showMessage() {
    alert( 'Hello everyone!' );
}
```

```
showMessage();
showMessage();
```

```
function name(parameter1, parameter2, ... parameterN) {
    ...
}
```

L'appel `showMessage()` exécute le code de la fonction. Ici, nous verrons le message deux fois, parce qu'on l'appelle deux fois.

Cet exemple illustre clairement l'un des principaux objectifs des fonctions: éviter la duplication de code.

Fonctions

mohamed@goumih.com

70

Variables locales: Une variable déclarée à l'intérieur d'une fonction n'est visible qu'à l'intérieur de cette fonction.

```
function showMessage() {  
    let message = "Hello, I'm JavaScript!"; // variable locale  
  
    alert( message );  
}  
showMessage(); // Hello, I'm JavaScript!  
  
alert( message ); // <-- Erreur! La variable est locale à la fonction
```

Variables externes : Une fonction peut également accéder à une variable externe, par exemple :

```
let userName = 'John';  
  
function showMessage() {  
    let message = 'Hello, ' + userName;  
    alert(message);  
}  
  
showMessage(); // Hello, John
```

```
let userName = 'John';  
  
function showMessage() {  
    userName = "Bob"; // (1) changé la variable externe  
  
    let message = 'Hello, ' + userName;  
    alert(message);  
}  
  
alert( userName ); // John avant l'appel de fonction  
  
showMessage();  
  
alert( userName ); // Bob, la valeur a été modifiée par la fonction
```

Fonctions :

71

Variables globales

Les variables déclarées en dehors de toute fonction, telle que **userName** externe dans le code ci-dessus, sont appelées *globales*.

Les variables globales sont visibles depuis n'importe quelle fonction (sauf si elles sont masquées par les variables locales). C'est une bonne pratique de minimiser l'utilisation de variables globales. Le code moderne a peu ou pas de variable globales.

La plupart des variables résident dans leurs fonctions. Parfois, cependant, ils peuvent être utiles pour stocker des données au niveau du projet.

- Un **paramètre** est la variable répertoriée entre parenthèses dans la fonction
- Un **argument** est la valeur qui est transmise à la fonction lorsqu'elle est appelée

Nous déclarons des fonctions en listant leurs paramètres, puis les appelons en passant des arguments.

Dans l'exemple ci-dessous on pourrait dire : "la fonction **showMessage** est déclarée avec deux paramètres, puis appelée avec deux arguments : **from** et "Hello".

```
function showMessage(from, text) { // arguments : from, text
    alert(from + ':' + text);
}
showMessage('Ann', 'Hello!'); // Ann: Hello! (*)
showMessage('Ann', "What's up?"); // Ann: What's up? (**)
```

Fonctions:

72

Si une fonction est appelée, mais qu'aucun argument n'est fourni, alors la valeur correspondante devient `undefined`. Par exemple, la fonction `showMessage(from, text)` mentionnée précédemment peut être appelée avec un seul argument :

```
showMessage("Ann");
```

Ce n'est pas une erreur. Un tel appel produirait "`*Ann*: undefined`". Comme la valeur de `text` n'est pas transmise, elle devient `undefined`.

Nous pouvons spécifier la valeur dite "**par défaut**" (à utiliser si omise) pour un paramètre dans la déclaration de fonction, en utilisant = :

```
function showMessage(from, text = "no text given") {  
    alert( from + ": " + text );  
}  
  
showMessage("Ann"); // Ann: aucun texte fourni
```

Maintenant, si le paramètre `text` n'est pas passé, il obtiendra la valeur "`no text given`".

Ici, "`no text given`" est une chaîne de caractères, mais il peut s'agir d'une expression plus complexe, qui n'est évaluée et affectée que si le paramètre est manquant. Donc, cela est également possible :

```
function showMessage(from, text = anotherFunction()) {  
    // anotherFunction() exécuté uniquement si aucun texte n'est fourni  
    // son résultat devient la valeur de text  
}
```

Fonctions: Renvoyer une valeur

73

Une fonction peut renvoyer une valeur dans avec la directive **return** peut être n'importe où dans la fonction. Lorsque l'exécution le permet, la fonction s'arrête et la valeur est renvoyée au code.

Il peut y avoir plusieurs occurrences de **return** dans une seule fonction.

Par exemple :

```
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}

let age = prompt('How old are you?', 18);

if (checkAge(age)) {
  alert('Access granted');
} else {
  alert('Access denied');
}
```

```
function sum(a, b) {
  return a + b;
}

let result = sum(1, 2);
alert(result); // 3
```

Il est possible d'utiliser **return** sans valeur. Cela entraîne la sortie immédiate de la fonction:

```
function showMovie(age) {
  if (!checkAge(age)) {
    return;
  }

  alert("Showing you the movie"); // (*)
  // ...
}
```

Fonctions: Nommer une fonction

74

Les fonctions sont des actions. Donc, leur nom est généralement un verbe. Il convient de décrire brièvement, mais aussi précisément que possible, le rôle de la fonction. Pour qu'une personne qui lit le code reçoive le bon indice. C'est une pratique répandue de commencer une fonction avec un préfixe verbal qui décrit vaguement l'action. Il doit exister un accord au sein de l'équipe sur la signification des préfixes. Par exemple, les fonctions qui commencent par "show" affichent généralement quelque chose.

Fonction commençant par...

- "get..." – retourne une valeur,
- "calc..." – calcule quelque chose,
- "create..." – créer quelque chose,
- "check..." – vérifie quelque chose et retourne un booléen, etc.

Les fonctions doivent être courtes et faire exactement une seule chose. Si cette chose est conséquente:

```
showMessage(...)      // affiche un message
getAge(...)          // renvoie l'âge (l'obtient en quelque sorte)
calcSum(...)         // calcule une somme et renvoie le résultat
createForm(...)      // crée un formulaire (et le retourne généralement)
checkPermission(...) // vérifie une permission, retourne vrai/faux
```

Fonctions: Résumé

75

Une déclaration de fonction ressemble à ceci :

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

- Les valeurs transmises à une fonction en tant que paramètres sont copiées dans ses variables locales.
 - Une fonction peut accéder à des variables externes. Mais cela ne fonctionne que de l'intérieur.
 - Le code en dehors de la fonction ne voit pas ses variables locales.
 - Une fonction peut renvoyer une valeur. Si ce n'est pas le cas, le résultat est `undefined`.
- Pour rendre le code propre et facile à comprendre,
il est recommandé d'utiliser principalement des variables et des paramètres locaux dans la fonction, et non des variables externes.
- Il est toujours plus facile de comprendre une fonction qui possède des paramètres, fonctionne avec eux et renvoie un résultat,
plutôt qu'une fonction qui ne comporte aucun paramètre,
mais modifie des variables externes comme un effet secondaireUn nom doit clairement décrire le rôle de la fonction.
Lorsque nous voyons un appel de fonction dans le code, un bon nom nous donne instantanément une compréhension
de ce qu'elle fait et de ce qu'elle retourne.
- Une fonction est une action, les noms de fonctions sont donc généralement verbaux.
 - Il existe de nombreux préfixes de fonctions bien connus, tels que `create...`, `show...`, `get...`, `check...` et ainsi de suite.
 - Utilisez-les pour indiquer ce que fait une fonction.

EP12:

76

1. Ecrire une fonction qui renvoie true si le paramètre age est supérieur à 18.

Sinon, il demande une confirmation et renvoie son résultat :

2. Ecrivez une fonction `min(a, b)` qui renvoie le plus petit des deux nombres a et b.

3. Ecrivez une fonction `pow(x, n)` qui renvoie x à la puissance n. Ou, autrement dit, multiplie x par lui-même n fois et renvoie le résultat.

Créez une demande (`prompt`) de x et n, puis affiche le résultat de `pow(x, n)`

4. Écrivez une fonction qui prend un nombre entier de minutes et le convertit en secondes.

5. Écrivez une fonction qui prend la base et la hauteur d'un triangle et retourne sa surface. Notez que la surface d'un triangle est: $(\text{base} * \text{hauteur}) / 2$

6. Écrivez un programme JavaScript pour trouver le plus grand des trois entiers donnés.

7. Créer une fonction qui calcule la table de multiplication d'un entier n:

Fonctions Expressions

77

Il existe une autre syntaxe pour créer une fonction appelée ***Expression de Fonction***. Cela nous permet de créer une nouvelle fonction au milieu de n'importe quelle expression. Par exemple :

```
let sayHi = function() {
    alert( "Hello" );
};
```

:quelle que soit la manière dont la fonction est créée, une fonction est une valeur. une fonction est stocké dans la variable sayHi.

La principale différence entre une demande réelle (ask) et l'exemple ci-dessus est que les fonctions réelles utilisent des moyens d'interagir avec l'utilisateur plus complexes que la simple confirmation (confirm). Dans le navigateur, une telle fonction dessine généralement une belle fenêtre de questions.

Les arguments showOk et showCancel de ask s'appellent des ***fonctions callback***

(fonctions de rappel) ou simplement des ***callbacks*** (rappels).

L'idée est que nous passions une fonction et attendons qu'elle soit "rappelée" plus tard si nécessaire. Dans notre cas, showOk devient le rappel pour la réponse "oui" et showCancel pour la "non" réponse.

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();
}

function showOk() {
    alert( "You agreed." );
}

function showCancel() {
    alert( "You canceled the execution." );
}
ask("Do you agree?", showOk, showCancel);
```

Fonctions fléchées :(lambda)

78

Il existe une syntaxe plus simple et concise pour créer des fonctions, c'est souvent mieux que les Expressions de Fonction.

Les "fonctions fléchées" sont appelées ainsi pour leur syntaxe :

```
let func = (arg1, arg2, ..., argN) => expression;
```

```
let sum = (a, b) => a + b;
```

/* Cette fonction fléchée est la forme raccourcie de :

```
let sum = function(a, b) {
  return a + b;
};
```

```
alert( sum(1, 2) ); // 3
```

```
let double = n => n * 2;
```

// Similaire à : let double = function(n) { return n * 2 }

```
alert( double(3) ); // 6
```

Les fonctions fléchées peuvent être utilisées de la même manière que les Expressions de Fonction.

Par exemple pour créer une fonction dynamiquement :

```
let age = prompt("What is your age?", 18);
```

```
let welcome = (age < 18) ?
  () => alert('Hello') :
  () => alert("Greetings!");
```

```
welcome(); // ok now
```

On peut utiliser les accolades et return dans les fonctions fléchées

```
let sum = (a, b) => { // Les accolades ouvre une fonction multiligne
  let result = a + b;
  return result; //
};

alert( sum(1, 2) ); // 3
```

EP13:

Remplacez les expressions de fonction par des fonctions fléchées dans le code ci-dessous :

```
function ask(question, yes, no) {
    if (confirm(question)) yes();
    else no();
}

ask(
    "Do you agree?",
    function() { alert("You agreed."); },
    function() { alert("You canceled the execution."); }
);
```

EP14: Dates

80

Ecrivez une fonction `getWeekDay(date)` pour afficher le jour de la semaine sous forme abrégée: ‘MO’, ‘TU’, ‘WE’, ‘TH’, ‘FR’, ‘SA’, ‘SU’.

des jours de la semaine commençant par lundi (numéro 1), puis mardi (numéro 2) et jusqu’au dimanche (numéro 7).

Ecrivez une fonction `getLocalDay(date)` qui renvoie le jour de la semaine “européen” pour `date`.

Ecrivez une fonction `getLastDayOfMonth(year, month)` qui renvoie le dernier jour du mois. Parfois, c'est 30, 31 ou même 28/29 février.

Paramètres:

- `year` – année à quatre chiffres, par exemple 2012.
- `month` – mois, de 0 à 11.

Par exemple, `getLastDayOfMonth(2012, 1)` = 29 (année bissextile, février).

Ecrivez une fonction `getSecondsToday()` qui renvoie le nombre de secondes depuis le début de la journée.

Par exemple, s'il est maintenant 10:00 am, et qu'il n'y a pas de décalage de l'heure d'été, alors :

La fonction devrait fonctionner dans n'importe quel jour. Autrement dit, il ne devrait pas avoir de valeur “aujourd’hui” codée en dur.

Fonction récursive

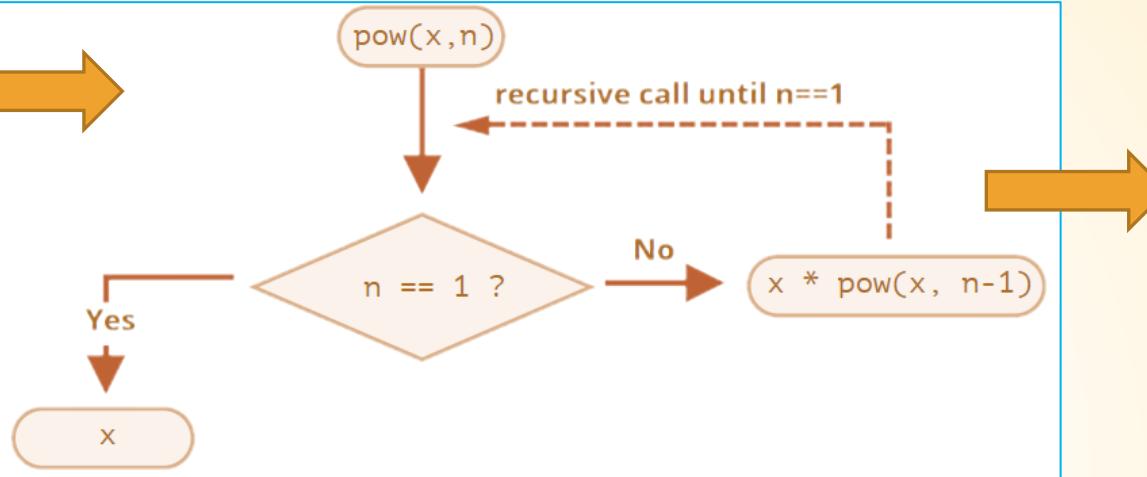
81

Exemple Fonction puissance

```
function pow(x, n) {
  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```



```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) ); // 8
```

Une solution récursive est généralement plus courte qu'une solution itérative.
Ici, nous pouvons réécrire la même chose en utilisant l'opérateur conditionnel ?
Au lieu de `if` pour rendre `pow (x, n)` plus concis et toujours très lisible:

Exemple Fonction Factorielle

```
function factorial(n) {
  return (n != 1) ? n * factorial(n - 1) : 1;
}
```

```
alert( factorial(5) ); // 120
```

```
function pow(x, n) {
  return (n == 1) ? x : (x * pow(x, n - 1));
}
```

EP15:

82

Écrire 2 fonction **somme(n)** (itérative et récursive) qui calcule la somme des nombres $1 + 2 + \dots + n$,

Par exemple :

somme(1) = 1 ,somme(2) = 2 + 1 = 3... somme(100) = 100 + 99 + ... + 2 + 1 = 5050

En mathématiques, la **suite de Fibonacci** est une suite d'entiers dans laquelle chaque terme est la somme des deux termes qui le précédent.

Notée (F_n) , elle est définie par $F_0 = 0$, $F_1 = 1$, et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$.

Les termes de cette suite sont appelés *nombres de Fibonacci* et forment la suite A000045 de l'OEIS :

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	...	F_n
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	...	$F_{n-1} + F_{n-2}$

Écrire une fonction itérative et récursive **fib(n)** qui retourne le Numéro de Fibonacci

setTimeout / setInterval

83

Peut-être que nous ne voulons pas exécuter une fonction tout de suite, mais à un certain moment dans le futur. Cela s'appelle "ordonnancer (ou planifier) un appel de fonction".

Il existe deux méthodes pour cela :

- **setTimeout** permet d'exécuter une fonction une unique fois après un certain laps de temps.
- **setInterval** nous permet d'exécuter une fonction de manière répétée, en commençant après l'intervalle de temps, puis en répétant continuellement à cet intervalle.

Ces méthodes ne font pas partie de la spécification JavaScript.

Mais la plupart des environnements ont un planificateur interne et fournissent ces méthodes. En particulier, elles sont supportées par tous les navigateurs et Node.js.

```
setTimeout(() => alert('Bonjour'), 1000);
```

Par exemple, le code ci-dessous appelle la fonction `sayHi()` une unique fois au bout de 1 seconde :

```
function sayHi() {
  alert('Hello');
}

setTimeout(sayHi, 1000);
```

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Bonjour", "Jean"); // Bonjour, Jean
```

```
// Se répète toutes les 2 secondes
let timerId = setInterval(() => alert('tick'), 2000);
```

```
// S'arrête après 5 secondes
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

La gestion des erreurs

84

il existe une construction de syntaxe `try...catch` qui permet au script "d'attraper" les erreurs et, au lieu de mourir en cas de problème, de faire quelque chose de plus raisonnable.

```
try {
    // code...
} catch (err) {
    // Gestion des erreurs
}
```

```
try {
    setTimeout(function() {
        noSuchVariable; // le script mourra ici
    }, 1000);
} catch (err) {
    alert( "won't work" );
}
```

Objet d'erreur: En cas d'erreur, JavaScript génère un objet contenant les détails à son sujet. L'objet est ensuite passé en argument à `catch`: toutes les erreurs intégrées, l'objet d'erreur a deux propriétés principales:

name

Nom de l'erreur. Par exemple, pour une variable non définie, il s'agit de "ReferenceError".

message

Message textuel sur les détails de l'erreur.

```
try {
    let user = JSON.parse(json); // <-- pas d'erreurs
    if (!user.name) {
        throw new SyntaxError("Incomplete data: no name"); // (*)
    }
    alert( user.name );
} catch (err) {
    alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data: no name
}
```

```
let error = new Error("Things happen o_0");
alert(error.name); // Error
alert(error.message); // Things happen o_0
```

try...catch...finally

85

La construction try...catch peut avoir une autre clause de code : finally.

S'il existe, il s'exécute dans tous les cas:

- après try, s'il n'y a pas eu d'erreur,
- après catch, s'il y a eu des erreurs.

Le code a deux manières d'exécution:

- 1.Si vous répondez "Yes" à "Make an error?", Alors try -> catch -> finally.
- 2.Si vous dites "No", alors try -> finally.

```
try {
  alert( 'try' );
  if (confirm('Make an error?')) BAD_CODE();
} catch (err) {
  alert( 'catch' );
} finally {
  alert( 'finally' );
}
```

```
let num = +prompt("Enter a positive integer number?", 35)

let diff, result;

function fib(n) {
  if (n < 0 || Math.trunc(n) != n) {
    throw new Error("Must not be negative, and also an integer.");
  }
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}
```

```
let start = Date.now();

try {
  result = fib(num);
} catch (err) {
  result = 0;
} finally {
  diff = Date.now() - start;
}

alert(result || "error occurred");

alert(`execution took ${diff}ms`);
```

TP2:

86

Créez un programme JavaScript qui calcule l'aire et le périmètre d'un rectangle en fonction des entrées de l'utilisateur. Le programme doit demander à l'utilisateur d'entrer la longueur et la largeur du rectangle à l'aide de la fonction **prompt**. Une fois que l'utilisateur a saisi la longueur et la largeur, le programme doit calculer et afficher l'aire et le périmètre du rectangle à l'aide de deux fonctions distinctes : **calculateArea()** et **calculatePerimeter()**.

Après avoir calculé et affiché l'aire et le périmètre du rectangle, demandez à l'utilisateur s'il souhaite calculer l'aire et le périmètre d'un autre rectangle. Si l'utilisateur répond "oui", répétez le processus depuis le début. Si l'utilisateur répond "non", terminez le programme.

Créez un programme JavaScript qui calcule le coût total d'une commande de pizza en fonction des entrées de l'utilisateur. Le programme doit demander à l'utilisateur d'entrer la taille de la pizza, le nombre de garnitures et s'il souhaite ou non être livré, à l'aide de la fonction **prompt**. Une fois que l'utilisateur a saisi ces informations, le programme doit calculer et afficher le coût total de la commande de pizza à l'aide de trois fonctions distinctes : **calculatePizzaCost()**, **calculateToppingsCost()** et **calculateDeliveryCost()**.

La fonction **calculatePizzaCost()** doit prendre la taille de la pizza comme argument et renvoyer le coût de la pizza. Le coût d'une pizza est calculé en fonction de sa taille comme suit : Petite pizza : 30DH Pizza moyenne 50DH ,Grande pizza :70DH.

La fonction **calculateToppingsCost()** doit prendre le nombre de garnitures comme argument et renvoyer le coût des garnitures. Le coût des garnitures est calculé comme suit : 0 garniture : 0DH 1 à 3 garnitures : 10DH par garniture , 4-6 garnitures : 8DH par garniture ,7+ garnitures : 6DH par garniture.

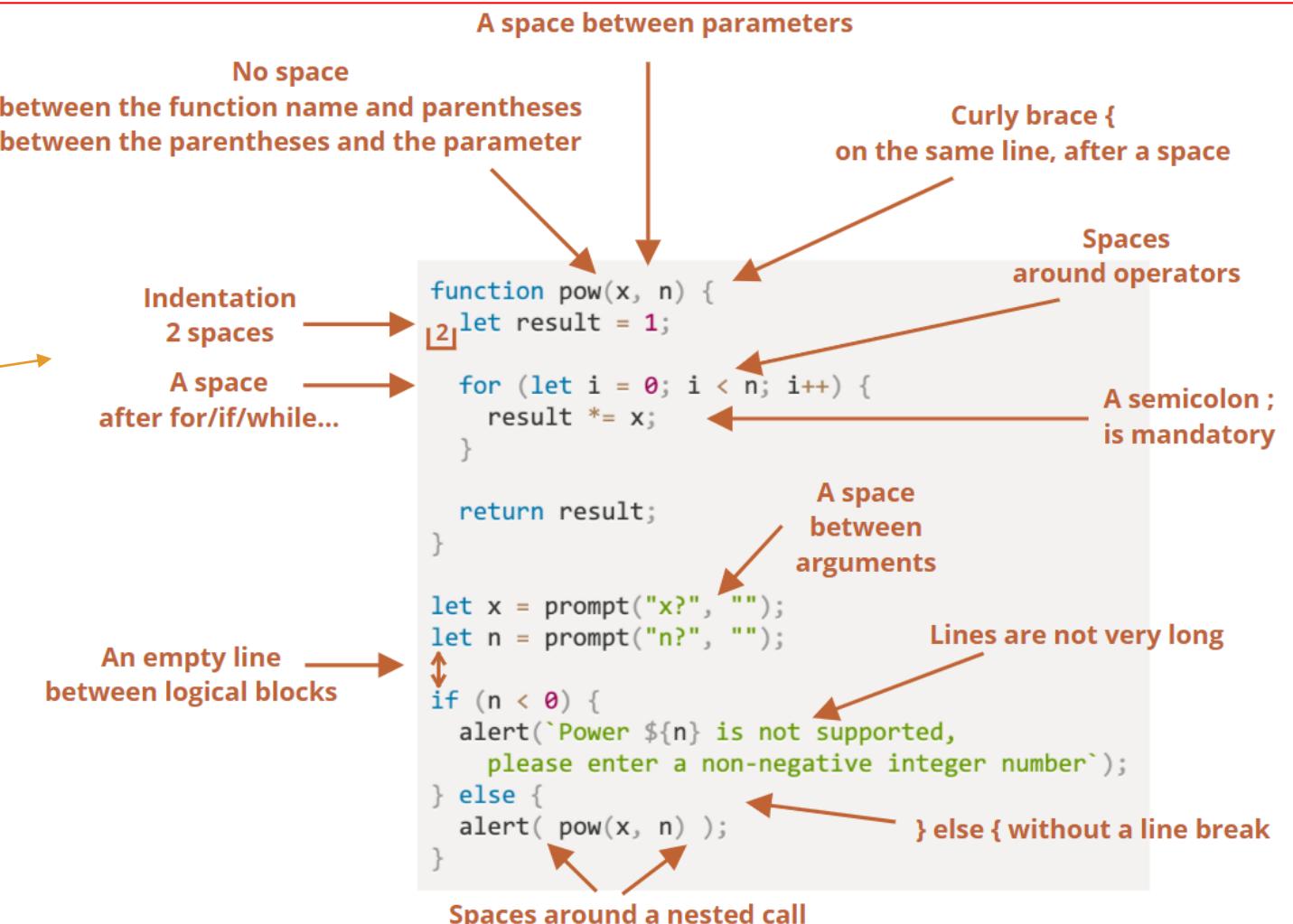
La fonction **calculateDeliveryCost()** doit prendre l'option de livraison comme argument et renvoyer le coût de la livraison. Le coût de la livraison est calculé comme suit : Cueillette : 0DH, Livraison : 8DH

Style de codage

PLUS 87

- Notre code doit être aussi propre et lisible que possible.
- C'est en fait un art de la programmation – prendre une tâche complexe et la coder de manière correcte et lisible par l'homme. Un bon style de code aide grandement à cela. Une chose à aider est le bon style de code.

Voici un aide-mémoire avec quelques règles suggérées (plus de détails ci-dessous) :



Quelques règles du code

PLUS 88

Accolades

Dans la plupart des projets JavaScript, les accolades sont écrites sur la même ligne que le mot clé correspondant, et non sur la nouvelle ligne, dans un style dit «égyptien». Il y a aussi un espace avant un crochet d'ouverture.

Comme ceci :

1. 😞 Les débutants font parfois cela. C'est une mauvaise pratique! Les accolades ne sont pas nécessaires : `if (n < 0) {alert(`Power ${n} is not supported`);}`
2. 😞 Lorsque vous n'utilisez pas d'accolades, évitez de passer pas à la ligne, il est facile de se tromper! : `if (n < 0)
 alert(`Power ${n} is not supported`);`
3. 😊 Ne pas utiliser d'accolade sur une seule ligne, est acceptable tant que cela reste court :
`if (n < 0) alert(`Power ${n} is not supported`);`
4. 😊 Voici une bonne manière de faire `if (n < 0) {
 alert(`Power ${n} is not supported`);
}`

```
if (condition) {  
    // fait ceci  
    // ...et cela  
    // ...et cela  
}
```

```
// les guillemets backtick ` permettent de scinder la chaîne de caractères en plus  
let str =  
  `ECMA International's TC39 is a group of JavaScript developers,  
  implementers, academics, and more, collaborating with the community  
  to maintain and evolve the definition of JavaScript.  
`;
```

```
if (  
    id === 123 &&  
    moonPhase === 'Waning Gibbous' &&  
    zodiacSign === 'Libra'  
) {  
    letTheSorceryBegin();  
}
```

Indentations

mohamed@goumih.com

PLUS 89

Il existe deux types d'indentations :

- **Un retrait horizontal : 2(4) espaces.**

Une indentation horizontale est faite en utilisant 2 ou 4 espaces ou le symbole horizontal de tabulation (key Tab). Lequel choisir est une vieille guerre sainte. Les espaces sont plus communs de nos jours.

Un des avantages des espaces sur les tabulations est qu'ils permettent des configurations de retrait plus flexibles que le symbole tabulation.

Par exemple, nous pouvons aligner les arguments avec le crochet d'ouverture, comme ceci :

- **Un retrait vertical: lignes vides pour fractionner le code en blocs logiques.**

Même une seule fonction peut souvent être divisée en blocs logiques. Dans l'exemple ci-dessous, l'initialisation des variables, la boucle principale et le retour du résultat sont fractionnés verticalement :

Des références pour mieux organiser le code :

- [Google JavaScript Style Guide](#)
- [Airbnb JavaScript Style Guide](#)
- [Idiomatic.JS](#)
- [StandardJS](#)

```
show(parameters,  
      aligned, // 5 espaces à gauche  
      one,  
      after,  
      another  
) {  
// ...  
}
```

```
function pow(x, n) {  
  let result = 1;  
  //           <--  
  for (let i = 0; i < n; i++) {  
    result *= x;  
  }  
  //           <--  
  return result;  
}
```

Voici quelques linters bien connus :

- [JSLint](#) – l'un des premiers linters.
- [JSHint](#) – plus de paramètres que
- [ESLint](#) – probablement le plus récent

Ninja code

PLUS 90

- ▶ Les programmeurs ninjas du passé ont utilisé ces astuces pour aiguiser l'esprit des mainteneurs de code.
- ▶ Les gourous de la révision de code les recherchent dans les tâches de test.
- ▶ Les développeurs novices les utilisent parfois encore mieux que les programmeurs ninjas.
- ▶ Lisez-les attentivement et découvrez qui vous êtes: un ninja, un novice ou peut-être un critique de code ?
- ▶ Faites le code aussi court que possible. Montrez à quel point vous êtes intelligent.

```
// tiré d'une bibliothèque javascript bien connue  
i = i ? i < 0 ? Math.max(0, len + i) : i : 0;
```

Une autre façon de coder plus rapidement consiste à utiliser des noms de variable d'une seule lettre partout. Comme a, b ou c.

Une petite variable disparaît dans le code comme un vrai ninja dans la forêt. Personne ne pourra la trouver en utilisant la "recherche" de l'éditeur. Et même si quelqu'un le fait, il ne pourra pas "déchiffrer" la signification du nom a ou b.

... Mais il y a une exception. Un vrai ninja n'utilisera jamais i comme compteur dans une boucle "for". N'importe où, mais pas ici. Regardez autour de vous, il y a beaucoup plus de lettres exotiques. Par exemple, x ou y.

Une variable exotique en tant que compteur de boucle est particulièrement intéressante si le corps de la boucle nécessite 1 à 2 pages (rallongez-la si vous le pouvez). Ensuite, si quelqu'un regarde au fond de la boucle, il ne sera pas en mesure de comprendre rapidement que la variable nommée x est le compteur de boucles.

- list → lst .

- userAgent → ua .

- browser → brsr .

- ...etc

Utiliser des abréviations

Partie 3 : les objets

- Objets**
- Boucle for ..in**
- Méthodes objets**
- le mot this**
- fonction constructeur**
- Méthode Call**

- Méthode Apply**
- Méthode Bind**
- Type symbol**
- JSON**
- JSON.stringify**
- JSON.parse**

Primitives vs Object

92

Une primitive : Est une valeur de type primitif.

- Il existe 7 types primitifs

: `string, number, bigint, boolean, symbol, null et undefined.`

Un objet :

- Est capable de stocker plusieurs valeurs en tant que propriétés.
- Peut être créé avec {}, par exemple: `{name: "John", age: 30}`.
- Il existe d'autres types d'objets en JavaScript : les fonctions, par exemple, sont des objets..

L'une des meilleures choses à propos des objets est que nous pouvons stocker une fonction en tant que l'une de ses propriétés.

Objets:

93

les objets sont utilisés pour stocker des collections de données variées et d'entités plus complexes.

En JavaScript, les objets pénètrent dans presque tous les aspects du langage

Un objet peut être créé avec des accolades `{...}`, avec une liste optionnelle de *propriétés*, Cette déclaration s'appelle un **littéral objet (object literal)**.

Ou avec la classe `Object()` :`new Object()`

```
let user = new Object(); // syntaxe "constructeur d'objet"
let user = {}; // syntaxe "littéral objet"
```

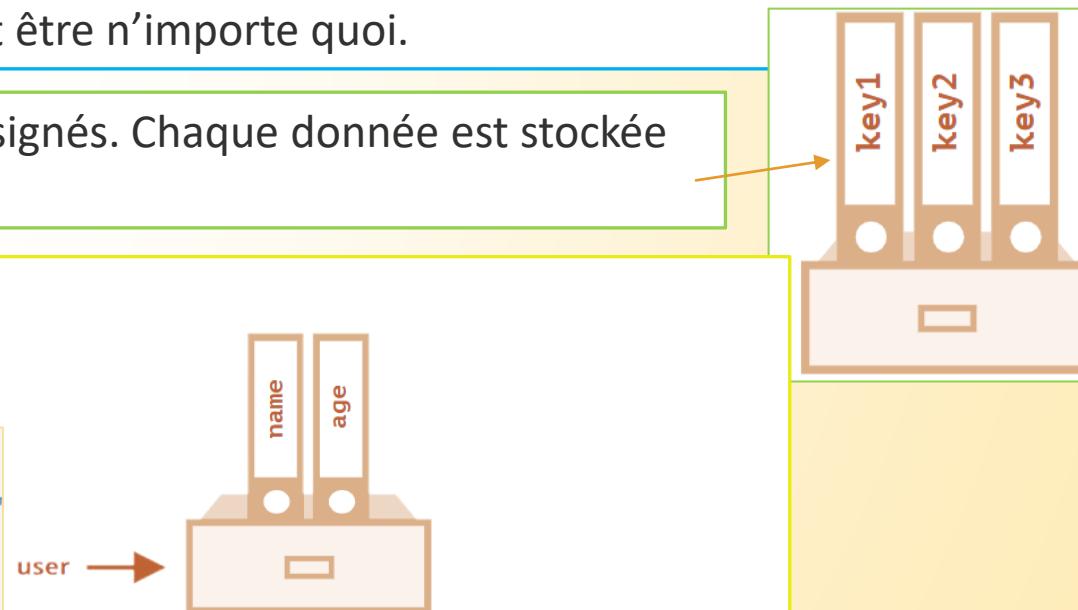
Une **propriété** est une paire “**clé: valeur**”, dans laquelle **la clé (key)** est une chaîne de caractères (également appelée “**nom de la propriété**”), **et la valeur (value)** peut être n’importe quoi.

Nous pouvons imaginer un objet comme une armoire avec des fichiers signés. Chaque donnée est stockée dans son fichier par la clé.

Dans l'objet `user`, il y a **deux propriétés** :

- 1.La première propriété porte le nom "name" et la valeur "John".
- 2.La seconde a le nom "age" et la valeur 30.

```
let user = { // un objet
  name: "John", // par clé "nom" valeur de stockage "John"
  age: 30 // par clé "age" valeur de stockage 30
};
```



L'objet `user` résultant peut être imaginé comme une armoire avec deux fichiers signés intitulés “nom” et “âge”.

Objets

94

Les valeurs de propriété sont accessibles à l'aide de la notation par points .

```
// récupère les valeurs de propriété de l'objet :  
alert( user.name ); // John  
alert( user.age ); // 30
```

La valeur peut être de tout type. Ajoutons un booléen : `user.isAdmin = true;`

Pour supprimer une propriété, nous pouvons utiliser l'opérateur `delete` : `delete user.age;`

Propriété multi mots:

```
let user = {  
  name: "John",  
  age: 30,  
  "likes birds": true // le nom de la propriété multi-mots doit être entourée  
};
```

Il existe une autre “notation entre crochets” qui fonctionne avec n’importe quelle chaîne :

```
let user = {};  
  
// set  
user["likes birds"] = true;  
  
// get  
alert(user["likes birds"]); // true  
  
// delete  
delete user["likes birds"];
```

Objets

95

Nous pouvons utiliser des crochets dans un objet littéral, lorsqu'on crée un objet. Cela s'appelle des **propriétés calculées (computed property)**.

nous utilisons souvent des variables existantes en tant que valeurs pour les noms de propriétés.

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {
  [fruit]: 5, // le nom de la propriété est tiré de la variable fruit
};

alert( bag.apple ); // 5 si fruit="apple"
```

Pour Vérifier l'existence d'une propriété dans un objet ,on utilise **in** :

```
let user = { name: "John", age: 30 };

alert( "age" in user ); // true, user.age existe
alert( "blabla" in user ); // false, user.blabla n'existe pas
```

Pour parcourir toutes les clés d'un objet, on utilise la boucle : **for..in**.
Meme chose pour les valeurs des clés

```
let user = {
  name: "John",
  age: 30,
  isAdmin: true
};

for(let key in user) {
  // keys
  alert( key ); // name, age, isAdmin
  // valeurs pour les clés
  alert( user[key] ); // John, 30, true
}
```

Objets :Boucle for ..in

```
let user = {
    name: "John",
    surname: "Smith"
};
user.age = 25; // Ajouter une clé de plus

// les propriétés non-entiers sont listées dans l'ordre de création
for (let prop in user) {
    alert( prop ); // name, surname, age
}
```

On veut afficher les clés dans l'ordre de la création

```
let codes = {
    "49": "Germany",
    "41": "Switzerland",
    "44": "Great Britain",
    // ...
    "1": "USA"
};

for(let code in codes) {
    alert(code); // 1, 41, 44, 49
}
```

pour résoudre le problème avec les indicatifs de téléphone, nous pouvons rendre ces indicatifs non entiers.
Ajouter un signe plus "+" avant chaque indicatif suffit.

```
let codes = {
    "+49": "Germany",
    "+41": "Switzerland",
    "+44": "Great Britain",
    // ...
    "+1": "USA"
};

for(let code in codes) {
    alert( +code ); // 49, 41, 44, 1
}
```

EP16:

97

Écrivez le code, une ligne pour chaque action :

- 1.Créer un objet vide user.
- 2.Ajoutez la propriété name avec la valeur John.
- 3.Ajoutez la propriété surname avec la valeur Smith.
- 4.Changer la valeur de name pour Votre nom.
- 5.Supprimez la propriété name de l'objet.

Créez une fonction `multiplyNumeric(obj)` qui multiplie toutes les valeurs de propriétés numériques de `obj` par 2.
Utilisez `typeof` pour tester si un `number`

```
// before the call
let menu = {
  width: 200,
  height: 300,
  title: "My menu"
};
```

```
multiplyNumeric(menu);
```

```
// after the call
menu = {
  width: 400,
  height: 600,
  title: "My menu"
};
```

Nous avons un objet stockant les salaires de notre équipe :

Écrivez le code pour additionner tous les salaires et les enregistrer dans la variable `sum`.

Devrait être égale à 390 dans l'exemple ci-dessus.

Si `salaries` est vide, le résultat doit être 0

```
let salaries = {
  mohamed: 100,
  Adam: 160,
  Eve: 130
};
```

Méthodes d'objet:

98

apprenons à user à dire bonjour :

```
let user = {
  // ...
};

// d'abord, déclarer
function sayHi() {
  alert("Hello!");
};

// puis ajouter comme une méthode
user.sayHi = sayHi;

user.sayHi(); // Hello!
```

```
let user = {
  name: "John",
  age: 30
};

user.sayHi = function() {
  alert("Hello!");
};

user.sayHi(); // Hello!
```

Ici, nous venons d'utiliser une fonction **expression** pour créer la fonction et l'affecter à la propriété **user.sayHi** de l'objet. Ensuite, nous pouvons l'appeler comme **user.sayHi()**. L'utilisateur peut maintenant parler! Une fonction qui est la propriété d'un objet s'appelle sa **méthode**. Nous avons donc ici une méthode **sayHi** de l'objet **user**.

Programmation orientée objet

Lorsque nous écrivons notre code en utilisant des objets pour représenter des entités, cela s'appelle une programmation orientée objet, en bref : "POO".

La programmation orientée objet est un élément important, une science intéressante en soi.

this

99

Il existe une syntaxe plus courte pour les méthodes dans un littéral d'objet :

```
// ces objets font la même chose
user = {
  sayHi: function() {
    alert("Hello");
  }
};

// la méthode abrégée semble mieux, non ?
user = {
  sayHi() { // identique à "sayHi: function(){...}"
    alert("Hello");
  }
};
```

En JavaScript, le mot clé **this** se comporte différemment de la plupart des autres langages de programmation. Il peut être utilisé dans n'importe quelle fonction, même si ce n'est pas une méthode d'un objet.

Il n'y a pas d'erreur de syntaxe dans le code suivant :

```
function sayHi() {
  alert( this.name );
}
```

"this" dans les méthodes

Il est courant qu'une méthode d'objet ait besoin d'accéder aux informations stockées dans l'objet pour effectuer son travail.

Par exemple, le code à l'intérieur de `user.sayHi()` peut nécessiter le nom de user.

Pour accéder à l'objet, une méthode peut utiliser le mot-clé this.

La valeur de `this` est l'objet "avant le point", celui utilisé pour appeler la méthode.

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    // "this" is the "current object"
    alert(this.name);
  }
};

user.sayHi(); // John
```

fonction constructeur

100

Les fonctions du constructeur sont techniquement des fonctions habituelles. Il existe cependant deux conventions :

1. Elles sont nommées avec une **lettre majuscule en premier**.
2. Elles ne devraient être exécutées qu'avec l'opérateur "**new**".

Quand une fonction est exécutée avec **new**, elle effectue les étapes suivantes :

1. Un nouvel objet vide est créé et affecté à **this**.
2. Le corps de la fonction est exécuté. Habituellement, il modifie **this**, y ajoutant de nouvelles propriétés.
3. La valeur de **this** est retournée.

En d'autres termes, **new User(...)** fait quelque chose comme :

```
function User(name) {
  // this = {} (implicitement)

  // ajoute des propriétés à this
  this.name = name;
  this.isAdmin = false;

  // return this (implicitement)
}
```

Maintenant, si nous voulons créer d'autres utilisateurs,
nous pouvons appeler **new User("Ann")**, **new User("Alice")** etc.
Beaucoup plus court que d'écrire littéralement à chaque fois, et aussi facile à lire.

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

```
function makeUser(name, age) {
  return {
    name: name,
    age: age,
    // ...autres propriétés
  };
}

let user = makeUser("John", 30);
alert(user.name); // John
```

fonction constructeur

mohamed@goumih.com

101

Si nous avons beaucoup de lignes de code concernant la création d'un seul objet complexe, nous pouvons les envelopper dans une fonction constructeur, comme ceci :

```
// create a function and immediately call it with new
let user = new function() {
    this.name = "John";
    this.isAdmin = false;

    // ...autre code pour la création d'utilisateur
    // peut-être une logique complexe et des déclarations
    // variables locales etc
};
```

L'utilisation de fonctions de constructeur pour créer des objets offre une grande flexibilité.

La fonction constructeur peut avoir des paramètres qui définissent comment construire l'objet et ce qu'il doit y mettre.

Bien sûr, nous pouvons ajouter à `this` non seulement des propriétés, mais également des méthodes.

Par exemple, `new User(name)` ci-dessous créer un objet avec le `name` donné et la méthode `sayHi` :

```
function User(name) {
    this.name = name;

    this.sayHi = function() {
        alert( "My name is: " + this.name );
    };
}

let john = new User("John");

john.sayHi(); // My name is: John

/*
john = {
    name: "John",
    sayHi: function() { ... }
}
*/
```

Méthode Call

102

Il existe une méthode de fonction intégrée spéciale `func.call(context, ...args)` qui permet d'appeler explicitement une fonction en définissant `this`.

La syntaxe est la suivante:

`func.call(context, arg1, arg2, ...)`

Par exemple, dans le code ci-dessous, nous appelons `sayHi` dans le contexte de différents objets:

`sayHi.call(user)` exécute `sayHi` fournissant `this = user`, et la ligne suivante définit `this = admin`:

```
function sayHi() {
  alert(this.name);
}

let user = { name: "John" };
let admin = { name: "Admin" };

// utilisons call pour passer différents objets en tant que "this"
sayHi.call( user ); // John
sayHi.call( admin ); // Admin
```

Et ici, nous utilisons `call` pour appeler `say` avec le contexte et la phrase donnés:

```
function say(phrase) {
  alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// user devient this, et "Hello" devient le premier argument
say.call( user, "Hello" ); // John: Hello
```

Méthode Apply

Au lieu de `func.call(this, ...arguments)`, nous pourrions utiliser `func.apply(this, arguments)`. La syntaxe de la méthode intégrée `func.apply` est :

Il exécute la fonction en définissant `this=context` et en utilisant un objet de type tableau `args` comme liste d'arguments. La seule différence de syntaxe entre `call` et `apply` est que `call` attend une liste d'arguments, tandis que `apply` prend un objet de type tableau avec eux. Donc ces deux appels sont presque équivalents :

```
const numbers = [5, 6, 2, 3, 7];
const max = Math.max.apply(null, numbers);
console.log(max); // 7

const min = Math.min.apply(null, numbers);
console.log(min); // 2
```

Méthode Bind()

104

Avec la méthode **bind()**, un objet peut emprunter une méthode à un autre objet. L'exemple ci-dessous crée 2 objets (**person** et **member**). L'objet **member** emprunte la méthode **fullname** à l'objet **person** :

```
const person = {  
    firstName:"John",  
    lastName: "Doe",  
    fullName: function() {  
        return this.firstName + " " + this.lastName;  
    }  
}  
  
const member = {  
    firstName:"ahmed",  
    lastName: "Mohamed",  
}  
  
let fullName = person.fullName.bind(member);  
  
console.log(fullName()); ahmed Mohamed
```

Méthode Bind()

105

Parfois, la méthode **bind ()** doit être utilisée pour éviter de perdre **this**. Dans l'exemple suivant, l'objet **person2** a une méthode **display**. Dans la méthode **display**, cela fait référence à l'objet **person2** :

Lorsqu'une fonction est utilisée comme rappel, **this** est perdue. Cet exemple essaiera d'afficher le nom de la personne après 3 secondes, mais il affichera **undefined** à la place :

```
const person2 = {
  firstName:"John",
  lastName: "Doe",
  display: function () {
    console.log(this.firstName + " " + this.lastName);  undefined undefined
  }
}
setTimeout(person2.display, 3000);
```

La méthode **bind()** résout ce problème :

```
const person3 = {
  firstName:"John",
  lastName: "Doe",
  display: function () {
    console.log(this.firstName + " " + this.lastName);  John Doe
  }
}
let display = person3.display.bind(person3);
setTimeout(display, 3000);
```

Exemple: call(),apply(),bind()

106

```
let stg={  
    id:1,  
    prenom:"ahmed",  
    afficherPrenom:function(){  
        console.log(this.prenom);  ahmed, mohamed, mohamed  
    }  
}  
let stg2={  
    id:2,  
    prenom:"mohamed",  
}  
let affich2=function(filiere,groupe){  
    console.log(`id :${this.id} -prenom: ${this.prenom}  
-Filiere :${filiere} -groupe  
: ${groupe}`)  
}  
stg.afficherPrenom();  
stg.afficherPrenom.call(stg2);  
stg.afficherPrenom.apply(stg2);
```

```
//!ajouter les arguments  
affich2.call(stg,"dev digital","dev101");  
affich2.apply(stg,[ "dev digital","dev101"]);  
  
//!Bind methode  
let copyStg=affich2.bind(stg2,"dev digital","dev102");  
  
copyStg();  
  
//!bind en generale exemple  
print=console.log.bind();  
print("hello")
```

EP17:

Créez un objet **calculator** avec trois méthodes utiliser this :

- **read()** demande deux valeurs et les enregistre en tant que propriétés d'objet.
- **sum()** renvoie la somme des valeurs sauvegardées.
- **mul()** multiplie les valeurs sauvegardées et renvoie le résultat.

```
let calculator = {
    // ... votre code ...
};

calculator.read();
alert( calculator.sum() );
alert( calculator.mul() );
```

Écrivez une fonction constructeur appelée **Person** qui prend deux arguments : **name** et **age**. La fonction doit créer un nouvel objet avec les propriétés **name** et **age**, et une méthode appelée **introduction** qui enregistre la chaîne "**Bonjour, je m'appelle [nom] et j'ai [âge] ans.**" à la console.

Utilisez la méthode **call()** pour transmettre le nouvel objet en tant que **this** à la fonction **Person**.

Écrivez une fonction appelée **sumArray** qui prend un tableau de nombres comme argument et renvoie la somme de tous les nombres. Utilisez la méthode **apply()** pour passer le tableau en tant qu'arguments à la méthode **reduce()**. Refaire la même question en appelant la méthode **call()**.

Type Symbol

108

Un “**Symbol**” représente un identifiant unique.

Une valeur de ce type peut être créée en utilisant **Symbol()** :

```
// id est un nouveau symbole
let id = Symbol();
```

Lors de la création, nous pouvons donner à symbole une description (également appelée **nom de symbol**).

```
// id est un symbole avec la description "id"
let id = Symbol("id");
```

Les symboles sont garantis d'être uniques. Même si nous créons beaucoup de symboles avec la même description, ce sont des valeurs différentes. La description est juste une étiquette qui n'affecte rien:

```
let id1 = Symbol("id");
let id2 = Symbol("id");

alert(id1 == id2); // false
```

Si nous voulons vraiment afficher un symbole, nous devons appeler **.toString()** dessus, comme ici :

```
let id = Symbol("id");
alert(id.toString()); // Symbol(id), maintenant ça marche
```

Ou récupérer la propriété **symbol.description** pour afficher la description uniquement :

```
let id = Symbol("id");
alert(id.description); // id
```

Type Symbol

109

Les symboles nous permettent de créer des propriétés “cachées” d'un objet, qu'aucune autre partie du code ne peut accéder accidentellement ou écraser.

Par exemple, si nous travaillons avec des objets user qui appartiennent à un code tiers, nous aimerais leur ajouter des identificateurs.

Utilisons une clé symbole pour cela :

```
let id = Symbol("id");
let user = {
  name: "John",
  age: 30,
  [id]: 123
};

for (let key in user) alert(key); // name, age (pas de symboles)

// l'accès direct par le symbole fonctionne
alert( "Direct: " + user[id] );
```

```
let user = { // belongs to another code
  name: "John"
};

let id = Symbol("id");

user[id] = 1;

alert( user[id] ); // nous pouvons accéder aux données
```

JSON

mohamed@goumih.com

110

Supposons que nous avons un objet complexe et que nous aimerais le convertir en chaîne, l'envoyer par le réseau ou simplement le rendre (l'output) à des fins de journalisation.

Naturellement, une telle chaîne devrait inclure toutes les propriétés importantes.

Nous pourrions implémenter la conversion comme ceci:

Le JSON (JavaScript Object Notation) est un format général pour représenter les valeurs et les objets.

Initialement, il était conçu pour JavaScript, mais de nombreux autres langages disposent également de bibliothèques pour le gérer.

JSON prend en charge les types de données suivants :

- **Objets { ... }, Tableaux [...], Primitives**

- **JavaScript fournit des méthodes:**

- **JSON.stringify** pour convertir des objets en JSON.

- **JSON.parse** pour reconvertis JSON en objet.

Par exemple, nous allons JSON.stringify un student (etudiant):

La méthode **JSON.stringify(student)** prend l'objet et le convertit en une chaîne.

La chaîne json résultante est appelé un objet *JSON-encoded* ou *serialized* ou *stringified* ou *marshalled*. Nous sommes prêts à l'envoyer par le câble ou à le placer dans un simple stockage de données.

```
let student = {
    name: 'John',
    age: 30,
    isAdmin: false,
    courses: ['html', 'css', 'js'],
    wife: null
};

let json = JSON.stringify(student);
alert(typeof json); // nous avons une string!

alert(json);
/* JSON-encoded object:
{
    "name": "John",
    "age": 30,
    "isAdmin": false,
    "courses": ["html", "css", "js"],
    "wife": null
}
*/
```

Json.stringify

JSON.stringify peut aussi être appliqué aux primitives.

```
// un nombre en JSON est juste un nombre
alert( JSON.stringify(1) ) // 1

// une chaîne en JSON est toujours une chaîne, mais entre guillemets
alert( JSON.stringify('test') ) // "test"

alert( JSON.stringify(true) ); // true

alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

JSON est une spécification indépendante du langage et ne contenant que des données.

Par conséquent, certaines propriétés d'objet spécifiques à JavaScript sont ignorées par **JSON.stringify**.

À savoir:

- Propriétés de fonction (méthodes).
- Clés et valeurs symboliques.
- Propriétés qui stockent `undefined`.

Le grand avantage est que les objets imbriqués sont pris en charge et convertis automatiquement.

```
let meetup = {
  title: "Conference",
  room: {
    number: 23,
    participants: ["john", "ann"]
  }
};

alert( JSON.stringify(meetup) );
```

```
let user = {
  sayHi() { // ignorée
    alert("Hello");
  },
  [Symbol("id")]: 123, // ignorée
  something: undefined // ignorée
};

alert( JSON.stringify(user) ); // {} (objet vide)
```

```
/* La structure entière est stringified:
{
  "title": "Conference",
  "room": {"number": 23, "participants": ["john", "ann"]},
}
*/
```

Json.parse :

convertir JSON en objet

```
let userData = '{ "name": "John", "age": 35,
  "isAdmin": false, "friends": [0,1,2,3] }';

let user = JSON.parse(userData);

document.write( user.friends[1] ); // 1
```

```
let str = '{"title": "Conference", "date": "2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str);

alert( meetup.date.getDate() ); // Error!
```

Utilisation de reviver

```
let str = '{"title": "Conference", "date": "2017-11-30T12:00:00.000Z"}';

let meetup = JSON.parse(str, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( meetup.date.getDate() ); // ça fonctionne maintenant!
```

```
let schedule = `{
  "meetups": [
    {"title": "Conference", "date": "2017-11-30T12:00:00.000Z"},
    {"title": "Birthday", "date": "2017-04-18T12:00:00.000Z"}
  ]
};

schedule = JSON.parse(schedule, function(key, value) {
  if (key == 'date') return new Date(value);
  return value;
});

alert( schedule.meetups[1].date.getDate() ); // ça fonctionne!
```

TP3:

113

1. Créez un objet appelé "**voiture**" avec les propriétés suivantes : Marque, modèle, année, couleur ,kilométrage .
2. Ensuite, ajoutez une méthode à l'objet « **voiture**" appelée "drive" qui accepte un nombre de KM comme argument et met à jour la propriété " **kilométrage** ".
3. Créez deux instances de l'objet "**voiture**", l'une représentant une Tesla Model S rouge 2020 avec 1000 km, et une autre représentant une Ford Mustang bleue 2015 avec 50000 km.
4. Appelez la méthode "drive" sur les deux instances, en passant 100 km pour la Tesla et 500 km pour la Mustang, et enregistrez la propriété "kilométrage" mise à jour pour chaque objet.
5. Utiliser une boucle pour imprimer toutes les propriétés de l'objet "voiture".
6. Utiliser "new" pour créer une nouvelle instance de l'objet « **voiture**" avec un ensemble de propriétés différents.
7. Utiliser l'opérateur "in" pour vérifier si une propriété existe dans l'objet "car" .
8. Créer un symbole unique à utiliser comme propriété de l'objet "voiture".
9. Ajouter une méthode `toString()` qui affiche les données des voitures.
10. Créer l'objet Voiture et ses méthodes avec la fonction constructeur.
11. Transformez l'objet **voiture** en JSON puis relisez-le dans une autre variable.

Partie 4 :

Nombres et Strings

- Nombres**
- Arrondir les nombres**
- Nombres: isNaN, isFinite**
- méthodes mathématiques**

- String :les caractères spéciaux**
- Méthodes de String**
- Comparer les String**

Nombres

115

Ici l'underscore _ joue le rôle de "sucre syntaxique", il rend le nombre plus lisible.

Le moteur JavaScript ignore simplement _ entre les chiffres, donc c'est exactement le même milliard que ci-dessus.

Imaginez que nous ayons besoin d'écrire 1 milliard. Le moyen évident est:

```
1 let milliard = 1000000000;
```

Nous pouvons également utiliser l'underscore _ comme séparateur :

```
1 let billion = 1_000_000_000;
```

En JavaScript, nous pouvons raccourcir un nombre en y ajoutant la lettre "e" et en spécifiant le nombre de zéros :

```
1 let milliards = 1e9; // 1 milliard, littéralement: 1 et 9 zéros
2
3 alert( 7.3e9 ); // 7.3 milliards (pareil que 7,300,000,000 ou 7_300_000_000)
```



En d'autres termes, e multiplie le nombre de 1 avec le nombre de zéros donné.

```
1 1e3 === 1 * 1000 // e3 signifie *1000
2 1.23e6 === 1.23 * 1000000 // e6 signifie *1000000
```

Nombres

116

Maintenant, écrivons quelque chose de très petit. Disons, 1 microseconde (un millionième de seconde):

```
1 let mcs = 0.000001;
```

Comme avant, l'utilisation de "e" peut nous aider. Si nous voulons éviter d'écrire les zéros explicitement, nous pourrions dire la même chose avec :

```
1 let mcs = 1e-6; // six zéros à gauche de 1
```

Les nombres Hexadécimaux sont souvent utilisés en JavaScript pour représenter des couleurs, encoder des caractères et pour beaucoup d'autres choses. Alors, naturellement, il existe un moyen plus court de les écrire: 0x puis le nombre.

```
1 alert( 0xff ); // 255  
2 alert( 0xFF ); // 255 (même résultat car la casse n'a pas d'importance)
```

Les systèmes numériques binaires et octaux sont rarement utilisés, mais sont également supportés avec les préfixes 0b et 0o :

```
1 let a = 0b11111111; // forme binaire de 255  
2 let b = 0o377; // forme octale de 255  
3  
4 alert( a == b ); // true, car a et b valent le même nombre 255
```

Nombres :**toString**

117

La méthode **num.toString(base)** retourne une chaîne de caractères représentant num dans le système numérique de la base donnée.

```
let num = 255;  
  
alert( num.toString(16) ); // ff  
alert( num.toString(2) ); // 11111111
```

La base peut varier de 2 à 36. Par défaut, il s'agit de 10.

Les cas d'utilisation courants sont:

- **base=16** est utilisé pour les couleurs hexadécimales, les encodages de caractères, etc. les chiffres peuvent être 0..9 ou A..F.
- **base=2** est principalement utilisé pour le débogage d'opérations binaires, les chiffres pouvant être 0 ou 1.
- **base=36** est le maximum, les chiffres peuvent être 0..9 ou A..Z. L'alphabet latin entier est utilisé pour représenter un nombre.
- Un cas amusant mais utile pour la base 36 consiste à transformer un identifiant numérique long en quelque chose de plus court, par exemple pour créer une URL courte. On peut simplement le représenter dans le système numérique avec base 36:

```
alert( 123456..toString(36) ); // 2n9c
```

noter que deux points sur **123456..toString(36)** n'est pas une faute de frappe. Si nous voulons appeler une méthode directement sur un nombre, comme **toString** dans l'exemple ci-dessus, nous devons placer deux points .. avant la méthode.

Nombres :Arrondir

118

Il existe plusieurs fonctions intégrées pour arrondir:

Math.floor

Arrondis: 3.1 devient 3, et -1.1 devient -2.

Math.ceil

Arrondis: 3.1 devient 4, et -1.1 devient -1.

Math.round

Arrondit à l'entier le plus proche: 3,1 devient 3, 3,6 devient 4, le cas du milieu : 3,5 arrondit également à 4.

Math.trunc (non pris en charge par Internet Explorer)

Supprime tout ce qui suit le point décimal sans avoir arrondi: 3.1 devient 3, -1.1 devient -1

Par exemple, nous avons 1.2345 et voulons l'arrondir à 2 chiffres, pour obtenir seulement 1.23.

Il y a deux façons de le faire:

Par exemple, pour arrondir le nombre au deuxième chiffre après la décimale, nous pouvons multiplier le nombre par 100 (ou une plus grande puissance de 10), appeler la fonction d'arrondi puis la diviser:

```
let num = 1.23456;

alert( Math.round(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // 0.30
```

```
let sum = 0.1 + 0.2;
alert( +sum.toFixed(2) ); // 0.3
```

Nombres :`isFinite` et `isNaN`

119

`isNaN(valeur)` convertit son argument en un nombre puis le teste pour qu'il soit NaN:

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

`isFinite(valeur)` convertit son argument en un nombre et renvoie true s'il s'agit d'un nombre régulier,

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, car c'est une valeur non régulière: NaN
alert( isFinite(Infinity) ); // false, car c'est aussi une valeur non régulière

let num = +prompt("Entrez un nombre", '');

// sera vrai, sauf si vous entrez Infinity, -Infinity ou NaN
alert( isFinite(num) );

alert( +"100px" ); // NaN
```

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5
```

```
alert( parseInt('12.3') ); // 12, seule la partie entière est renvoyée
alert( parseFloat('12.3.4') ); // 12.3, le deuxième point arrête la lecture
```

Il y a des situations où `parseInt` / `parseFloat` retournera `NaN`. Cela arrive quand on ne peut lire aucun chiffre:

parseInt et parseFloat

La conversion numérique à l'aide d'un plus + ou `Number()` est strict. Si une valeur n'est pas exactement un nombre, elle échoue:

```
1 alert( parseInt('a123') ); // NaN, le premier symbole arrête le processus
```

Nombres :méthodes mathématiques

120

JavaScript a un objet **Math** intégré qui contient une petite bibliothèque de fonctions et de constantes mathématiques.

Quelques exemples:

Math.random()

Retourne un nombre aléatoire de 0 à 1 (1 non compris)

```
alert( Math.random() ); // 0.1234567894322  
alert( Math.random() ); // 0.5435252343232  
alert( Math.random() ); // ... (tout nombre aléatoire)
```

Math.max(a, b, c...) / Math.min(a, b, c...)

Retourne le plus grand / le plus petit des nombres indiqués en argument.

```
alert( Math.max(3, 5, -10, 0, 1) ); // 5  
alert( Math.min(1, 2) ); // 1
```

Math.pow(n, power)

Renvoie n élevé à la puissance power donnée.

```
alert( Math.pow(2, 10) ); // 2 puissance 10 = 1024
```

Nombres :Résumé

121

Pour écrire de nombres avec beaucoup de zéros :Ajoutez "e" avec le nombre de zéros au nombre comme: 123e6 est 123 avec 6 zéros soit 123000000.

Un nombre négatif après le "e" entraîne la division du nombre par 1 avec des zéros donnés. Comme: 123-e6 est 123

Il est possible d'écrire des nombres directement dans les systèmes hex (0x), octal(0o) et binaire (0b).

parseInt(str,base) passe la chaîne de caractères str vers un système numérique avec une base donnée : $2 \leq \text{base} \leq 36$.

num.toString(base) convertit un nombre en chaîne de caractères dans le système numérique de la base donnée.

Utiliser parseInt/parseFloat pour la conversion des entiers et réels

Arrondissez en utilisant **Math.floor**, **Math.ceil**, **Math.trunc**, **Math.round** ou **num.toFixed(précision)**.

Voir l'objet **Math** quand vous en avez besoin. La bibliothèque est très petite, mais peut couvrir les besoins de base.

Créez une fonction **readNumber** qui invite à entrer un nombre jusqu'à ce que le visiteur saisisse une valeur numérique valide.

La valeur résultante doit être renvoyée sous forme de nombre.

Le visiteur peut également arrêter le processus en entrant une ligne vide ou en appuyant sur "CANCEL".

Dans ce cas, la fonction doit renvoyer null.

Ecrivez une boucle infinie avec For & While

Créez une fonction **randomInteger(min, max)** qui génère un nombre entier aléatoire compris entre min et max (min et max inclus).

Tout nombre compris dans l'intervalle min .. max doit apparaître avec la même probabilité.

TP4:

123

Créer une calculatrice qui permet de saisir 2 nombres et un opérateur :les opérations sont les méthodes vu dans le cours :

1. num1 et num2 doivent des nombres valides .
2. Ensuite, nous invitons l'utilisateur à effectuer l'opération et utilisons une instruction switch .
3. Pour l'opération de division (/), nous devons vérifier si le deuxième nombre est fini..
4. Pour l'opération de racine carrée (sqrt), nous vérifions si le premier nombre est négatif.
5. Nous utilisons également les méthodes Math.floor(), Math.ceil(), Math.trunc() et Math.round() pour effectuer des opérations de plancher, de plafond, de troncature et d'arrondi sur le premier nombre, respectivement.

créez une calculatrice de remboursement de prêt: Vous devez calculer le paiement mensuel, le total des intérêts payés et le montant total payé sur la durée du prêt. Pour ce faire :

1. vous devez d'abord utiliser la méthode parseFloat() pour convertir les valeurs d'entrée en nombres.
2. Ensuite, vous pouvez utiliser la méthodetoFixed() pour formater les nombres de sortie à deux décimales.
3. Vous utiliserez la méthode Math.pow() pour calculer l'exposant nécessaire pour déterminer le montant du paiement mensuel.
4. Ensuite, vous utiliserez la méthode Math.round() pour arrondir le montant du paiement mensuel au centime le plus proche.
5. Pour calculer le total des intérêts payés, vous devez soustraire le principal du prêt du montant total payé et utiliser à nouveau la méthodetoFixed() pour formater le nombre de sortie.
6. Enfin, vous utiliserez la méthode isNaN() pour vérifier les valeurs d'entrée non valides.

String: les caractères spéciaux

```
alert( "\u00A9" ); // ©
alert( "\u{20331}" ); // 僑, un rare hiéroglyphe chinois (long unicode)
alert( "\u{1F60D}" ); // 😊, un symbole de visage souriant (un autre long
```

Caractère	Description
\n	Nouvelle ligne
\r	Dans les fichiers texte Windows, une combinaison de deux caractères \r\n représente une nouvelle pause, tandis que sur un système d'exploitation non Windows, il s'agit simplement de \n . C'est pour des raisons historiques, la plupart des logiciels Windows comprennent également \n .
\' , \"	Quotes
\\\	Backslash
\t	Tab
\b , \f , \v	Backspace, Form Feed, Vertical Tab – conservés pour compatibilité, non utilisés de nos jours.
\xXX	Caractère Unicode avec l'unicode hexadécimal donné XX , par exemple '\x7A' est le même que 'z' .
\uXXXX	Un symbole Unicode avec le code hexadécimal XXXX en encodage UTF-16, par exemple \u00A9 – est un unicode pour le symbole de copyright © . Ce doit être exactement 4 chiffres hexadécimaux.
\u{X...XXXXXX} (de 1 à 6 caractères hexadécimaux)	Un symbole Unicode avec l'encodage UTF-32. Certains caractères rares sont encodés avec deux symboles Unicode, prenant 4 octets. De cette façon, nous pouvons insérer des codes longs.

String /Méthodes des String

125

Tous les caractères spéciaux commencent par un backslash (barre oblique inversée) \. On l'appelle aussi "caractère d'échappement"

```
alert(`I'm the Walrus!`); // I'm the Walrus!
```

```
alert('I\'m the Walrus!'); // I'm the Walrus!
```

```
alert(`The backslash: \\`); // The backslash: \
```

La propriété **length** a la longueur de la chaîne de caractères : `alert(`My\n`.length); // 3`

Pour obtenir un caractère à la position pos, utilisez des crochets **[pos]** ou appelez la méthode **str.charAt(pos)**

Le premier caractère commence à la position zéro :

```
let str = 'Hi';

str = 'h' + str[1]; // remplace la haine de caractères

alert(str); // hi
```

```
let str = `Hello`;

// le premier caractère
alert(str[0]); // H
alert(str.at(0)); // H

// le dernier caractère
alert(str[str.length - 1]); // o
alert(str.at(-1));
```

```
let str = `Hello`;

// le premier caractère
alert(str[0]); // H
alert(str.charAt(0)); // H

// le dernier caractère
alert(str[str.length - 1]); // o
```

Nous pouvons également parcourir les caractères en utilisant un **for..of** :

```
for (let char of "Hello") {
  alert(char); // H,e,l,l,o (char devient "H", ensuite "e", ensuite "l" etc)
}
```

Méthodes de String

126

Les méthodes **toLowerCase()** et **toUpperCase()** modifient la casse :

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
```

Il existe plusieurs façons de rechercher une partie d'une chaîne de caractères.

La première méthode est **str.indexOf(substr, pos)**.

Il cherche le **substr** dans **str**, en partant de la position donnée **pos**, et retourne la position où la correspondance a été trouvée ou -1 si rien ne peut être trouvé.

Par exemple :

```
let str = 'Widget with id';

alert( str.indexOf('Widget') ); // 0, parce que 'Widget' est trouvé au début
alert( str.indexOf('widget') ); // -1, pas trouvé, la recherche est sensible à la casse

alert( str.indexOf("id") ); // 1, "id" est trouvé à la position 1 (..idget with id)
```

Le second paramètre optionnel nous permet de rechercher à partir de la position donnée.

Par exemple, la première occurrence de "id" est à la position 1. Pour rechercher l'occurrence suivante, commençons la recherche à partir de la position 2 :

```
let str = 'Widget with id';

alert( str.indexOf('id', 2) ) // 12
```

str.lastIndexOf(pos)

Il y a aussi une méthode similaire **str.lastIndexOf(pos)** qui cherche de la fin d'une chaîne de caractères à son début. Elle liste les occurrences dans l'ordre inverse.

Méthodes de String

La méthode **str.includes(substr, pos)** retourne true/false en fonction de si str contient substr:

```
alert( "Widget".includes("id") ); // true
alert( "Widget".includes("id", 3) ); // false, à partir de la position 3,
```

Les méthodes **str.startsWith** et **str.endsWith** font exactement la même chose:

```
alert( "Widget".startsWith("Wid") ); // true, "Widget" démarre avec "Wid"
alert( "Widget".endsWith("get") ); // true, "Widget" fini avec "get"
```

```
alert( "Widget with id".includes("Widget") ); // true
alert( "Hello".includes("Bye") ); // false
```

Il existe 3 méthodes en JavaScript pour faire **slicing (obtenir le substring :sous-chaine)**: **substring**, **substr** et **slice**:

```
let str = "stringify";
alert( str.slice(0, 5) ); // 'strin', le substring de 0 à 5 (sans inclure 5)
alert( str.slice(0, 1) ); // 's', de 0 à 1, mais sans inclure 1, donc 0 à 0
```

```
let str = "stringify";
alert( str.slice(2) ); // 'ringify', à partir de la 2e position
```

```
let str = "stringify";
// commence à la 4ème position à partir de la droite, se termine au 1er à partir de la droite
alert( str.slice(-4, -1) ); // 'gif'
```

```
let str = "stringify";
alert( str.substr(2, 4) ); // 'ring',
à partir de la 2ème position on obtient 4 caractères
```

méthodes	sélection ...	valeurs négatives
slice(start, end)	de start à end (n'inclue pas end)	permet les négatifs
substring(start, end)	entre start et end	les valeurs négatives signifient 0
substr(start, length)	de start obtient length caractères	permet un start négatif

```
// ce sont les mêmes pour substring
alert( str.substring(2, 6) ); // "ring"
alert( str.substring(6, 2) ); // "ring"

// ...mais pas pour slice :
alert( str.slice(2, 6) ); // "ring" (le même résultat)
alert( str.slice(6, 2) ); // "" (une chaîne de caractères vide)
```

Comparer les String

128

`str.codePointAt(pos):`

Renvoie un nombre décimal représentant le code du caractère à la position pos :

```
// différentes lettres majuscules ont des codes différents
alert( "Z".codePointAt(0) ); // 90
alert( "z".codePointAt(0) ); // 122
alert( "z".codePointAt(0).toString(16) ); // 7a (
```

String.fromCodePoint(code):Crée un caractère par son code chiffre

```
alert( String.fromCodePoint(90) ); // Z  
alert( String.fromCodePoint(0x5a) ); // Z
```

Exemple : les caractères avec les codes 65 .. 220 (l'alphabet latin et un peu plus):

Les caractères majuscules sont les premiers, puis quelques spéciaux, puis les minuscules, et Ö vers la fin de la sortie.
Maintenant, cela devient évident pourquoi a > Z.

comparaisons par :localCompare

une méthode spéciale pour comparer des chaînes de caractères dans différentes langues, en respectant leurs règles.

L'appel **str.localeCompare(str2)** renvoie un entier indiquant si **str** est inférieur, égal ou supérieur à **str2** selon les règles du langage :

- Renvoie un nombre négatif si **str** est inférieur à **str2**
- Renvoie un nombre positif si **str** est supérieur à **str2**
- Renvoie 0 s'ils sont équivalents.

```
console.log('ä'.localeCompare('z', 'de')); // une valeur négative : en allemand  
ä est avant z  
  
console.log('ä'.localeCompare('z', 'sv')); // une valeur positive : en suédois,  
ä arrive après z
```

Array: Résumé des String

130

Il existe 3 types de quotes. Les Backticks permettent à une chaîne de s'étendre sur plusieurs lignes et d'intégrer des expressions \${...}.

Les chaînes de caractères en JavaScript sont encodées en UTF-16.

Nous pouvons utiliser des caractères spéciaux comme \n et insérer des lettres par leur unicode en utilisant \u....

Pour obtenir un caractère, utilisez : [].

Pour obtenir un substring utilisez : slice ou substring.

Pour mettre une chaîne de caractères en minuscule ou en majuscule, utilisez : toLowerCase/toUpperCase.

Pour rechercher un substring utilisez : indexOf, ou includesstartsWith/endsWith pour de simple vérifications.

Pour comparer les chaînes de caractères en fonction de la langue, utilisez : localeCompare, sinon, ils sont comparés par les codes de caractères.

Il existe plusieurs autres méthodes utiles dans les strings :

str.trim() – retire les espaces (“trims”) du début et de la fin de la chaîne de caractères.

str.repeat(n) – répète la chaîne de caractères n fois.

... et plus. Voir le [manuel](#) pour plus de détails.

EP19:

131

Écrire une fonction **ucFirst(str)** qui retourne la chaîne de caractères **str** avec le premier caractère majuscule, par exemple :

Créer une fonction **truncate(str, maxlen)** qui vérifie la longueur de **str** et, si elle dépasse **maxlength** – remplace la fin de **str** avec le caractère des ellipses "...", rendre sa longueur égale à **maxlength**.
Le résultat de la fonction doit être la chaîne de caractères tronquée (si nécessaire).

Nous avons un coût sous la forme "\$120". C'est-à-dire que le signe dollar commence, puis le nombre.
Créer une fonction **extractCurrencyValue(str)** qui extrait la valeur numérique d'une telle chaîne de caractères et la renvoie.

créer un programme qui intègre une liste de mots de vocabulaire et permet à l'utilisateur d'effectuer diverses opérations sur les mots. Le programme doit inviter l'utilisateur à entrer une liste de mots de vocabulaire séparés par des virgules. Par exemple : "pomme, banane, cerise, datte".

Le programme doit stocker la liste des mots dans un tableau.

Le programme doit permettre à l'utilisateur d'effectuer les opérations suivantes sur la liste de mots :

1. Inverser l'ordre des mots dans la liste.
2. Triez les mots par ordre alphabétique.
3. Convertissez les mots en minuscules.
4. Convertissez les mots en majuscules.
5. Comptez le nombre d'occurrences d'un mot spécifique dans la liste.
6. Trouver l'index d'un mot spécifique dans la liste.
7. Remplacez un mot spécifique dans la liste par un nouveau mot.
8. Supprimer toutes les occurrences d'un mot spécifique de la liste.
9. Concaténer deux ou plusieurs listes de mots ensemble.

Le programme doit afficher les résultats de l'opération sélectionnée par l'utilisateur sur la console.

Pour implémenter le programme, vous devrez utiliser les méthodes suivantes pour les chaînes :

`.split() .join() .sort() .toLowerCase() .toUpperCase .indexOf() .lastIndexOf() .replace() .replaceAll()
.concat()`

Partie 5 :

Les Arrays

- tableau
- Boucles
- Boucle forEach
- multidimensionnel
- push/pop/shift/unshift
- splice
- slice(concat
- Rechercher avec : filter /find ..

- Trier avec :sort() /reverse(..
- split/join
- map
- reduce
- copywithin
- some /every
- Array.isArray

Array: les tableaux

134

Il existe une structure de données spéciale appelée **Array (Tableau)**, pour stocker les collections ordonnées.

- Il existe deux syntaxes pour créer un tableau vide:
- Pour supprimer un élément on utilise `delete arr[1];`
- Les éléments du tableau sont numérotés, et commence par zéro.
- On peut obtenir un élément par son numéro grâce aux crochets:

On peut modifier une valeur par l'indice :

```
fruits[2] = 'Pear'; // maintenant ["Apple", "Orange", "Pear"]
```

```
fruits[3] = 'Lemon'; // maintenant ["Apple", "Orange", "Pear", "Lemon"]
```

```
let arr = new Array();
let arr = [];
let arr = new Array("Apple", "Pear", "etc");
let fruits = ["Apple", "Orange", "Plum"]
```

```
let fruits = ["Apple", "Orange", "Plum"];
alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
```

- Un tableau peut stocker des éléments de tout type:

```
let fruits = ["Apple", "Orange", "Plum"]
```

```
alert( fruits.length ); // 3
```

```
let fruits = ["Apple", "Orange", "Plum"];
```

```
alert( fruits ); // Apple,Orange,Plum
```

```
// mélange de valeurs
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello'); } ];
```

```
// récupère l'objet à l'index 1 et montre ensuite son nom
alert( arr[1].name ); // John
```

```
// affiche la fonction à l'index 3 et l'exécute la
arr[3](); // hello
```

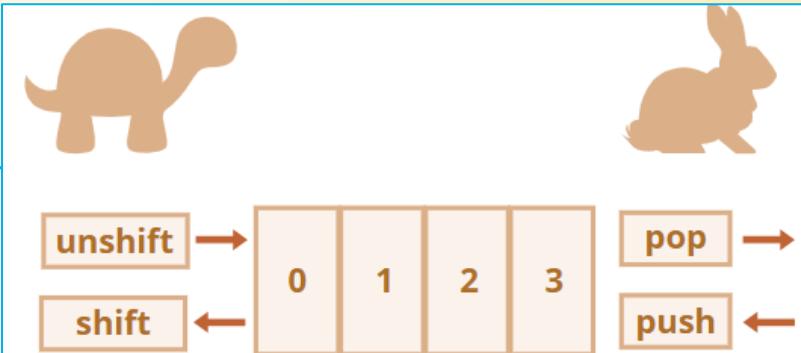
Pour obtenir la taille du tableau en utilisant la méthode `length`:

```
let fruits = [];
fruits[123] = "Apple";
alert( fruits.length ); // 124
```

Array :Boucles

135

Les méthodes **push/pop** vont vite, alors que **shift/unshift** sont lentes.



L'une des méthodes les plus anciennes pour cybler des éléments de tableau est la boucle **for** sur les **index**:

```
let arr = ["Apple", "Orange", "Pear"]

for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
```

il existe une autre forme de boucle pour boucler les tableaux, **for..of** :

```
let fruits = ["Apple", "Orange", "Plum"]

// itère sur des éléments de tableau
for (let fruit of fruits) {
  alert( fruit );
}
```

Techniquement, les tableaux sont des objets, il est également possible d'utiliser **for..in**:

```
let arr = ["Apple", "Orange", "Pear"]

for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
}
```

Array :Boucle forEach

136

La méthode **forEach** permet d'exécuter une fonction pour chaque élément du tableau.
La syntaxe:

```
arr.forEach(function(item, index, array) {  
    // ... fait quelques chose avec l'élément  
});
```

```
// pour chaque élément appelle l'alerte  
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {  
    alert(` ${item} est à l'index ${index} dans ${array}`);  
});
```

Array: String/ multidimensionnel

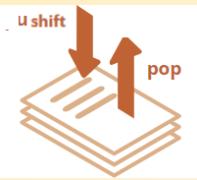
137

Les tableaux ont leur propre méthode **String** qui renvoie une liste d'éléments séparés par des virgules.

```
let arr = [1, 2, 3];  
  
alert( arr ); // 1,2,3  
alert( String(arr) === '1,2,3' ); // true
```

Déclaration d'un tableau multidimensionnel :

```
let matrix = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
];  
  
alert( matrix[1][1] ); // 5, l'élément central
```



Array : l'ajout et suppression



138

Les méthodes pop/push, shift/unshift

Une queue (file d'attente) est l'une des utilisations les plus courantes pour les tableaux. En informatique, cela signifie une collection ordonnée d'éléments qui supporte deux opérations :

- **push** ajoute un élément à la fin.
- **Unshift**: Ajoute l'élément au début du tableau:
- **Shift**: enlève un élément depuis le début, en faisant avancer la file d'attente, de sorte que le deuxième élément devienne le premier.
- **Pop** :enlève un élément de la fin

```
let fruits = ["Apple"];
fruits.push("Orange", "Peach");
fruits.unshift("Pineapple", "Lemon");
// ["Pineapple", "Lemon", "Apple", "Orange", "Peach"]
alert( fruits );
```

```
let fruits = ["Apple", "Orange"];
fruits.push("Pear");
alert( fruits ); // Apple, Orange, Pear
```



```
let fruits = ["Orange", "Pear"];
fruits.unshift('Apple');
alert( fruits ); // Apple, Orange, Pear
```



```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits.shift() ); // supprime Apple et l'alerte
alert( fruits ); // Orange, Pear
```



```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits.pop() ); // supprime "Pear" et l'alerte
alert( fruits ); // Apple, Orange
```

Les méthodes push et unshift peuvent ajouter plusieurs éléments à la fois:

Array: méthode splice

139

La méthode **splice** est un couteau suisse pour les tableaux. Elle peut tout faire : ajouter, supprimer et remplacer des éléments.

```
let arr = ["I", "study", "JavaScript", "right", "now"];

// supprime les 2 premiers éléments
let removed = arr.splice(0, 2);

alert( removed ); // "I", "study" <-- tableau des éléments supprimés
```

```
let arr = ["I", "study", "JavaScript"];

arr.splice(1, 1); // À partir de l'index 1 supprime 1 élément

alert( arr ); // ["I", "JavaScript"]
```

- les index négatifs sont autorisés dans **splice**. Ils spécifient la position à partir de la fin du tableau, comme ici:
- Supprimer et ajouter :

```
let arr = ["I", "study", "JavaScript"];

// de l'index 2
// supprime 0
// et ajoute "complex" et "language"
arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

```
let arr = [1, 2, 5];

// de l'index -1 (un déplacement à partir de la fin)
// supprime 0 éléments,
// puis insère 3 et 4
arr.splice(-1, 0, 3, 4);

alert( arr ); // 1,2,3,4,5
```

Array : méthodes slice & concat

140

Il retourne un nouveau tableau dans lequel il copie tous les éléments index qui commencent de start à end (sans compter end).

Les deux **start** et **end** peuvent être négatifs, dans ce cas, la position depuis la fin du tableau est supposée..

```
let arr = ["t", "e", "s", "t"];
alert( arr.slice(1, 3) ); // e,s (copy from 1 to 3)
alert( arr.slice(-2) ); // s,t (copy from -2 till the end)
```

La méthode **concat** crée un nouveau tableau qui inclut les valeurs d'autres tableaux et des éléments supplémentaires.

```
let arr = [1, 2];
// créer un tableau à partir de : arr et [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4
// créer un tableau à partir de : arr et [3,4] et [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6
// créer un tableau à partir de : arr et [3,4], puis ajoute les valeurs 5 et 6
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Recherche dans un tableau

141

Les méthodes `indexOf` `lastIndexOf` et `includes` ont la même syntaxe et utilisent essentiellement la même chose que leurs équivalents de chaîne, mais fonctionnent sur des éléments au lieu de caractères:

- `arr.indexOf(item, from)`: recherche l'élément `item` à partir de l'index `from`, et retourne l'index où il a été trouvé, sinon il retourne -1.
- `arr.lastIndexOf(item, from)` : pareil, mais regarde de droite à gauche.
- `arr.includes(item, from)` : recherche l'élément `item` en commençant par l'index `from`, retourne true si il est trouvé.

La méthode `find` recherche un seul (premier) élément qui rend la fonction true.

`findIndex` : recherche l'index du mot cherché

S'il y en a beaucoup plus, nous pouvons utiliser `filter()`

La syntaxe est à peu près identique à celle de `find`, mais `filter` renvoie un tableau d'éléments correspondants :

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

let user = users.find(item => item.id == 1);

alert(user.name); // John
```

```
let arr = [1, 0, false];

alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1

alert( arr.includes(1) ); // true ////////////// arreter ici
```

```
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];

// retourne les tableaux des deux premiers users
let someUsers = users.filter(item => item.id < 3);

alert(someUsers.length); // 2
```

Trier un tableau

142

La méthode **sort** trie le tableau *en place*, en changeant son ordre d'élément...

```
let arr = [ 1, 2, 15 ];
// la méthode réordonne le contenu de arr
arr.sort();

alert( arr ); // 1, 15, 2
```

Littéralement, tous les éléments sont convertis en chaînes de caractères pour comparaisons. Pour les chaînes de caractères, l'ordre lexicographique est appliqué et donc "2" > "15". Pour utiliser notre propre ordre de tri, nous devons fournir une fonction comme argument :

```
let arr = [ 1, 2, 15 ];

arr.sort(function(a, b) { return a - b; });
// Ou Avec fonction fléchée

alert(arr); // 1, 2, 15
```

localeCompare pour les chaînes de caractères

```
let countries = ['Österreich', 'Andorra', 'Vietnam'];
alert( countries.sort( (a, b) => a > b ? 1 : -1) ); // Andorra, Vietnam, Österreich

alert( countries.sort( (a, b) => a.localeCompare(b) ) ); // Andorra, Österreich, Vietnam
```

La méthode **reverse** inverse l'ordre des éléments dans l'arr.

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();

alert( arr ); // 5,4,3,2,1
```

Méthodes :split ,join

143

La méthode **split(delim)** fait exactement cela. Il divise la chaîne en un tableau par le délimiteur donné :

La méthode **split** a un deuxième argument numérique facultatif – une limite sur la longueur du tableau. En pratique, il est rarement utilisé :

L'appel de **split(s)** avec un s vide diviserait la chaîne en un tableau de lettres:

```
let names = 'Bilbo, Gandalf, Nazgul';

let arr = names.split(', ');

for (let name of arr) {
  alert(`Un message à ${name}.`); // Un message à Bilbo
}

let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(',', 2);

alert(arr); // Bilbo, Gandalf

let str = "test";

alert( str.split('') ); // t,e,s,t
```

L'appel de **join(séparateur)** fait l'inverse de **split**. Il crée une chaîne de caractères avec les éléments de arr fusionnés entre eux par séparateur.

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];

let str = arr.join(';'); // glue the array into a string using ;

alert( str ); // Bilbo;Gandalf;Nazgul
```

Array:Méthode map

144

La méthode **map()** crée un nouveau tableau avec les résultats de l'appel d'une fonction fournie sur chaque élément du tableau appelant ,voici des exemples :

```
var nombres = [1, 4, 9];
var doubles = nombres.map(function(num) {
  return num * 2;
});
// doubles vaut désormais [2, 8, 18].
// nombres vaut toujours [1, 4, 9]
```

```
var nombres = [1, 4, 9];
var racines = nombres.map(Math.sqrt);
// racines vaut désormais [1, 2, 3]
// nombres vaut toujours [1, 4, 9]
```

```
const array1 = [1, 4, 9, 16];
// pass a function to map
const map1 = array1.map(x => x * 2);

console.log(map1);
// résultat| Array [2, 8, 18, 32]
```

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length)
alert(lengths); // 5,7,6
```

Array: Méthode reduce

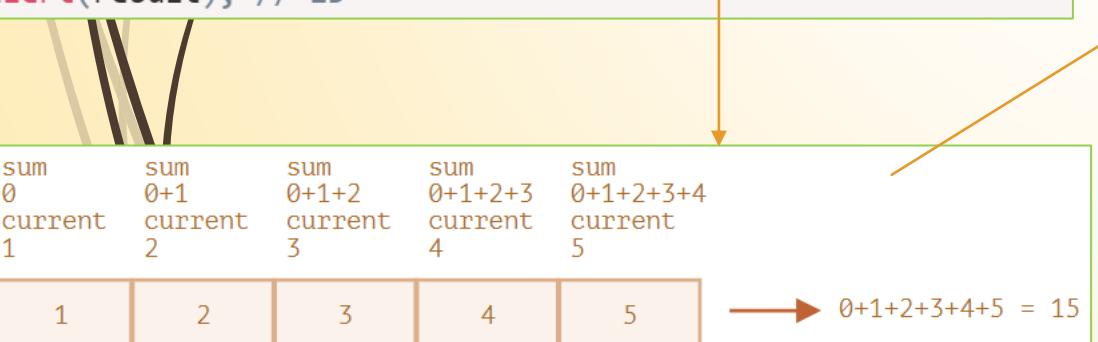
145

La méthode **reduce()** applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite), afin de la réduire à une seule valeur.
La syntaxe est la suivante:

```
let value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

- **accumulator** – est le résultat de l'appel de fonction précédent, égal à **initial** la première fois (si **initial** est fourni).
- **Item – valeurCourante** du tableau.
- **index** – est sa position.
- **array** – est le tableau.

```
let arr = [1, 2, 3, 4, 5];
let result = arr.reduce((sum, current) => sum + current, 0);
alert(result); // 15
```



```
sum=[0, 1, 2, 3, 4].reduce(function(accumulateur, valeurCourante, index, array){
  return accumulateur + valeurCourante;
});

alert(sum)
```

	sum	current	result
premier appel	0	1	1
deuxième appel	1	2	3
troisième appel	3	3	6
quatrième appel	6	4	10
cinquième appel	10	5	15

Méthode copywithin()

146

La méthode **copyWithin()** effectue une copie superficielle (*shallow copy*) d'une partie d'un tableau sur ce même tableau et le renvoie, sans modifier sa taille.

La syntaxe est la suivante :

```
arr.copyWithin(cible)
arr.copyWithin(cible, début)
arr.copyWithin(cible, début, fin)
```

```
const array1 = ['a', 'b', 'c', 'd', 'e'];

// copy to index 0 the element at index 3
console.log(array1.copyWithin(0, 3, 4));
// ["d", "b", "c", "d", "e"]

// copy to index 1 all elements from index 3 to the end
console.log(array1.copyWithin(1, 3));
// ["d", "d", "e", "d", "e"]
```

Array :Méthodes every et some

La méthode **every()** permet de tester si tous les éléments d'un tableau vérifient une condition donnée par une fonction en argument. Cette méthode renvoie true ou false.

```
const inf40 = (valeur) => valeur < 40;  
  
const array1 = [1, 30, 39, 29, 10, 13];  
  
console.log(array1.every(isBelowThreshold)); // true
```

La méthode **some()** teste si au moins un élément du tableau passe le test implémenté par la fonction fournie. Elle renvoie true et false.

```
const array = [1, 2, 3, 4, 5];  
  
// vérifié un nombre paire  
const even = (element) => element % 2 === 0;  
  
// vérifie au moins l'existence d'un nombre paire  
console.log(array.some(even)); // true
```

Méthode :`Array.isArray`

148

Les tableaux ne forment pas un type de langue distinct. Ils sont basés sur des objets.
Donc son `typeof` ne permet pas de distinguer un objet brut d'un tableau:

```
alert(typeof {}); // object  
alert(typeof []); // object (pareil)
```

`Array.isArray(value)` renvoie `true` si la `value` est un tableau, sinon il renvoie `false`.

```
alert(Array.isArray({})); // false  
alert(Array.isArray([])); // true
```

Résumé des méthodes des Arrays

149

Pour ajouter / supprimer des éléments :

- `push(...items)` – ajoute des éléments à la fin,
- `pop()` – extrait un élément en partant de la fin,
- `shift()` – extrait un élément depuis le début,
- `unshift(...items)` – ajoute des éléments au début.
- `splice(pos, deleteCount, ...items)` – à l'index pos supprime les éléments deleteCount et insert items.
- `slice(start, end)` – crée un nouveau tableau, y copie les éléments de start jusqu'à end (non inclus).
- `concat(...items)` – retourne un nouveau tableau: copie tous les membres du groupe actuel et lui ajoute des éléments. Si un des items est un tableau, ses éléments sont pris.

Pour rechercher parmi des éléments:

- `indexOf/lastIndexOf(item, pos)` – cherche l'item à partir de la position pos, retourne l'index -1 s'il n'est pas trouvé.
- `includes(value)` – retourne true si le tableau a une value, sinon false.
- `find/filter(func)` – filtrer les éléments à travers la fonction, retourne en premier / toutes les valeurs qui retournent true.
- `findIndex` est similaire à `find`, mais renvoie l'index au lieu d'une valeur.

Pour parcourir les éléments :

- `forEach(func)` – appelle func pour chaque élément, ne retourne rien.

Résumé des méthodes des Arrays(2)

150

Pour transformer le tableau:

- **map(func)** – crée un nouveau tableau à partir des résultats de func pour chaque élément.
- **sort(func)** – trie le tableau sur place, puis le renvoie.
- **reverse()** – inverse le tableau sur place, puis le renvoie.
- **split/join** – convertit une chaîne en tableau et retour.
- **reduce(func, initial)** – calcule une valeur unique sur le tableau en appelant func pour chaque élément et en transmettant un résultat intermédiaire entre les appels.

Aditionellement:

Array.isArray(value) vérifie que value est un tableau, si c'est le cas, renvoie true, sinon false.

Veuillez noter que les méthodes sort, reverse et splice modifient le tableau lui-même.

Ces méthodes sont les plus utilisées, elles couvrent 99% des cas d'utilisation.

Mais il y en a encore d'autres:

arr.some(ar) / arr.every(fn) vérifie le tableau.:

si fn renvoie une valeur vraie, arr.some() renvoie immédiatement true et arrête de parcourir les autres éléments ;

si fn renvoie une valeur fausse, arr.every()retourne immédiatement false et arrête également d'itérer sur les autres éléments.

Résumé des méthodes des Arrays(3)

mohamed@goumih.com

151

arr.fill(value, start, end) – remplit le tableau avec une répétition de value de l'index start à end.

arr.copyWithin(target, start, end) – copie ses éléments de la position start jusqu'à la position end into *itself*, à la position target (écrase les éléments existants).

arr.flat(depth)/arr.flatMap(fn) créer un nouveau tableau plat à partir d'un tableau multidimensionnel.

Pour la liste complète des méthodes, consultez le [manuel](#).

Essayons 5 opérations de tableau.

- 1.Créez un tableau **styles** avec les éléments “Ja” et “Blues”.
- 2.Ajoutez “RRoll” à la fin.
- 3.Remplacez la valeur au milieu par “Classiques”. Votre code pour trouver la valeur moyenne devrait fonctionner pour tous les tableaux de longueur impaire.
- 4.Extrayez la première valeur du tableau et affichez-la.
- 5.Ajoutez R et Reg au tableau.

Écrivez la fonction **sumInput()** qui:

- Demande à l'utilisateur des valeurs utilisant **prompt** et stocke les valeurs dans le tableau.
- Finit de demander lorsque l'utilisateur entre une valeur non numérique, une chaîne vide ou appuie sur “Annuler”.
- Calcule et retourne la somme des éléments du tableau.

P.S. Un zéro 0 est un nombre valide, donc s'il vous plaît n'arrêtez pas l'entrée sur zéro.

EP21:

153

L'entrée est un tableau de nombres, par ex. `arr = [1, -2, 3, 4, -9, 6]`.

La tâche est la suivante: trouver le sous-tableau contigu de `arr` avec la somme maximale des items.

Écrire la fonction `getMaxSubSum(arr)` qui retournera cette somme.

For instance:

```
1 getMaxSubSum([-1, 2, 3, -9]) == 5 (la somme des éléments en surbrillance)
2 getMaxSubSum([2, -1, 2, 3, -9]) == 6
3 getMaxSubSum([-1, 2, 3, -9, 11]) == 11
4 getMaxSubSum([-2, -1, 1, 2]) == 3
5 getMaxSubSum([100, -9, 2, -3, 5]) == 100
6 getMaxSubSum([1, 2, 3]) == 6 (prend tout)
```

Si tous les éléments sont négatifs, cela signifie que nous n'en prenons aucun (le sous-tableau est vide), la somme est donc zéro:

```
1 getMaxSubSum([-1, -2, -3]) = 0
```

154

Ecrivez la fonction **camelize(str)** qui change les mots séparés par des tirets comme “my-short-string” en camel-cased “myShortString”.

Donc: supprime tous les tirets et met en majuscule la première lettre de chaque mot à partir du deuxième mot.

```
camelize("background-color") == 'backgroundColor';
camelize("list-style-image") == 'listStyleImage';
camelize("-webkit-transition") == 'WebkitTransition';
```

utilisez **split** pour scinder la chaîne dans un tableau, transformer la et ensuite **join**.

Ecrivez une fonction **filterRangeInPlace(arr, a, b)** qui obtient un tableau arr et en supprime toutes les valeurs, sauf celles comprises entre a et b. Le test est: $a \leq arr[i] \leq b$.

La fonction doit juste modifier que le tableau. Elle ne doit rien retourner.

```
let arr = [5, 3, 8, 1];
filterRangeInPlace(arr, 1, 4);
alert( arr ); // [3, 1]
```

Nous avons un tableau de chaînes arr. Nous aimerais en avoir une copie triée, mais sans modifier arr.

Créez une fonction **copySorted(arr)** qui renvoie une copie triée.

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 28 };

let users = [ john, pete, mary ];

let names = users.map(item => item.name);

alert( names ); // John, Pete, Mary
```

```
let arr = ["HTML", "JavaScript", "CSS"];
let sorted = copySorted(arr);

alert( sorted ); // CSS, HTML, JavaScript
alert( arr ); // HTML, JavaScript, CSS (aucune modification)
```

EP23:

155

Ecrivez la fonction **shuffle(array)** qui mélange les éléments (de manière aléatoire) du tableau. Les exécutions multiples de shuffle peuvent conduire à différents ordres d'éléments. Par exemple:

```
let arr = [1, 2, 3];
shuffle(arr);
// arr = [3, 2, 1]
```

Ecrivez la fonction **getAverageAge(users)** qui obtient un tableau d'objets avec la propriété age et qui ensuite retourne l'age moyen. La formule pour la moyenne est $(\text{age1} + \text{age2} + \dots + \text{ageN}) / N$.

```
let john = { name: "John", age: 25 };
let pete = { name: "Pete", age: 30 };
let mary = { name: "Mary", age: 29 };

let arr = [ john, pete, mary ];

alert( getAverageAge(arr) ); // (25 + 30 + 29) / 3 = 28
```

Soit le tableau suivant :

```
const countries = [ { name: 'United States', code: 1 }, { name: 'Canada', code: 2 }, { name: 'United Kingdom', code: 3 }, { name: 'France', code: 4 }, { name: 'Germany', code: 5 }, { name: 'Morocco', code: 6 }];
```

1. Utiliser la méthode **reduce** pour calculer la somme des codes de tous les countries.
2. Utilisez la méthode **filter** pour créer un nouveau tableau contenant uniquement les countries dont le code est supérieur ou égal à 3 .
3. Utilisez **map** pour créer un nouveau tableau contenant uniquement les noms des pays .
4. utilisez **sort** pour trier les pays par code dans l'ordre décroissant .
5. utilisez **reduce** pour calculer la valeur totale des codes de tous les pays dont le nom commence par 'U' .
6. utilisez **push** pour ajouter un nouvel pays à la fin du tableau avec le nom "Japan" et le code 7 .
7. Utilisez **pop** pour supprimer le dernier objet country du tableau .
8. utilisez **splice** pour supprimer l'objet avec le code 4 du tableau .
9. utilisez **slice** pour créer un nouveau tableau contenant uniquement les 3 premiers objets country .
10. Utilisez **find** pour renvoyer le pays avec le code 2
11. Utilisez **some** pour vérifier s'il existe au moins un objet pays avec un code supérieur ou égal à 5
12. utilisez **every** pour vérifier si tous les objets country ont un nom d'au moins 3 caractères .

1. utilisez **includes** pour vérifier si le tableau contient un objet country avec le name "Canada".
2. utilisez **trim** pour supprimer tout espace autour des noms de pays .
3. utilisez **repeat** pour créer un nouveau tableau d'objets où le nom de chaque pays est répété 3 fois .
4. utilisez **localeCompare** pour trier le tableau alphabétiquement par nom de pays .
5. Utilisez **at** pour obtenir l'objet country à l'index 2 .
6. utilisez une boucle **for...of** pour afficher chaque nom de pays dans la console .
7. utilisez **concat** pour ajouter deux autres objets , 'Australia ' et 'china' au tableau .
8. utilisez **join** pour créer une chaîne qui répertorie les noms de pays séparés par une virgule .
9. Étant donné une chaîne qui répertorie plusieurs noms de pays séparés par une virgule
`const countryString = 'Italy, Spain, Portugal';`
 Utilisez **split** pour créer un tableau d'objets de pays .
 Et ajouter le au tableau **countries** .
1. utilisez **copyWithin** pour remplacer les objets country aux indices 1 et 2 par l'objet country à l'index 0 .
2. utilisez **shift** pour supprimer le premier objet country du tableau .
3. utilisez **unshift** pour ajouter un nouvel pays 'KSA' au début du tableau .
4. Utiliser la combinaison de la méthode **filter ,map et sort** ,pour déterminer les pays qui ont un code >5 trié par ordre décroissant.

Partie 6 : Les itérables

-Les itérables

-Map

-Set

-weakMap

-weakSet

-object.keys/values/entries

-Array.from

-object.fromEntries

-object et récursivité

-L'affection par décomposition

Itérables

158

Les objets ***Iterable*** sont une généralisation des tableaux.

C'est un concept qui permet de rendre n'importe quel objet utilisable dans une boucle **for .. of**.

Bien sûr, les tableaux sont itérables. Mais il existe de nombreux autres objets intégrés, qui sont également itérables.

Par exemple, les chaînes de caractères sont également itérables.

Si un objet n'est pas techniquement un tableau, mais représente une collection (liste, set) de quelque chose, alors **for .. of** est une excellente syntaxe pour boucler.

Les tableaux et les chaînes sont les itérables intégrés les plus largement utilisés.

```
for (let char of "test") {
    // se déclenche 4 fois: une fois pour chaque personnage
    alert( char ); // t, ensuite e, ensuite s, ensuite t
}
```

- ***Iterables*** sont des objets qui implémentent la méthode `Symbol.iterator`, comme décrit ci-dessus.
- ***Array-likes*** sont des objets qui ont des `index` et des `length`, ils ressemblent donc à des tableaux.

```
let arrayLike = { // a des index et une longueur => semblable à un tableau
    0: "Hello",
    1: "World",
    length: 2
};

// Erreur (pas de Symbol.iterator)
for (let item of arrayLike) {}
```

Map

159

Une **Map** (dictionnaire de donnée) permet, comme pour un **Object**, de stocker plusieurs éléments sous la forme de clés valeurs. Sauf que cette fois, les clés peuvent être de n'importe quelle type. Contrairement aux **Object**, Map conserve l'ordre d'insertion des valeurs.

Voici les méthodes et les propriétés d'une Map :

`new Map()` – instancie la map.

`map.set(key, value)` – définit la valeur pour une clé.

`map.get(key)` – retourne la valeur associée à la clé,

`map.has(key)` – retourne `true` si `key` existe, sinon `false`.

`map.delete(key)` – supprime la valeur associée à `key`

`map.clear()` – supprime tout le contenu de la map.

`map.size` – retourne le nombre d'éléments.

```
let map = new Map();

map.set('1', 'str1');    // une clé de type chaîne de
                        caractère
map.set(1, 'num1');     // une clé de type numérique
map.set(true, 'bool1'); // une clé de type booléenne

// souvenez-vous, dans un `Object`, les clés sont
// converties en chaîne de caractères
// alors que `Map` conserve le type d'origine de la
// clé,
// c'est pourquoi les deux appels suivants retournent
// des valeurs différentes:
```

```
alert( map.get(1) ); // 'num1'
alert( map.get('1') ); // 'str1'
```

```
alert( map.size ); // 3
```

Map

Parcourir les éléments d'une Map

Il existe 3 façons de parcourir les éléments d'une map :

- **map.keys()** – retourne toutes les clés sous forme d'iterable,
- **map.values()** – retourne les valeurs sous forme d'iterable,
- **map.entries()** – retourne les entries (couple sous forme de [clé, valeur]),
- c'est la méthode utilisée par défaut par **for..of**.

cucumber	tomatoes	onion
<hr/>		
500		
350		
50		
<hr/>		
cucumber,500	tomatoes,350	onion,50

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);

// on parcourt les clés (les légumes)
for (let vegetable of recipeMap.keys()) {
  document.write(vegetable+"<br>"); // cucumber,
                                         tomatoes, onion
}
document.write("<hr>")

// on parcourt les valeurs (les montants)
for (let amount of recipeMap.values()) {
  document.write(amount+"<br>"); // 500, 350, 50
}
document.write("<hr>")

// on parcourt les entries (couple [clé, valeur])
for (let entry of recipeMap) { // équivalent à :
  recipeMap.entries()
  document.write(entry); // cucumber,500 (etc...)
}
```

Map

161

Il est aussi possible d'utiliser `forEach` avec **Map** comme on pourrait le faire avec un tableau :

```
// exécute la fonction pour chaque couple (key, value)
recipeMap.forEach( (value, key, map) => {
  alert(` ${key}: ${value}`); // cucumber: 500 etc
});
```

Lorsqu'une Map est créée, nous pouvons passer un tableau (ou un autre itérable) avec des paires clé/valeur pour l'initialisation, comme ceci :

Si nous avons un objet simple et que nous aimerais créer une carte à partir de celui-ci, nous pouvons utiliser la méthode intégrée **Object.entries(obj)** qui renvoie un tableau de paires clé/valeur pour un objet exactement dans ce format. Nous pouvons donc créer une carte à partir d'un objet comme celui-ci :

```
let obj = {
  name: "John",
  age: 30
};

let map = new Map(Object.entries(obj));

alert( map.get('name') ); // John
```

```
// array of [key, value] pairs
let map = new Map([
  ['1', 'str1'],
  [1, 'num1'],
  [true, 'bool1']
]);

alert( map.get('1') ); // str1
```

la méthode **Object.fromEntries** qui fait l'inverse : étant donné un tableau de paires [clé, valeur], elle crée un objet à partir de celles-ci :

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);

// now prices = { banana: 1, orange: 2, meat: 4 }

alert(prices.orange); // 2
```

Set

162

Set est une liste sans doublons, Ses principales méthodes sont :

- `new Set(iterable)` – crée un `set`, si un `iterable` (la plupart du temps, un tableau) est passé en paramètre, ses valeurs sont copiées dans le `set`
- `set.add(value)` – ajoute l'élément `value` et retourne le `set`.
- `set.delete(value)` – supprime l'élément `value` et retourne `true` si la valeur existait au moment de l'appel sinon `false`.
- `set.has(value)` – retourne `true` si la valeur existe dans le `set`, sinon faux.
- `set.clear()` – supprime tout le contenu du `set`.
- `set.size` – le nombre d'éléments dans le tableau.

Ce qu'il faut surtout savoir c'est que lorsque l'on appelle plusieurs fois `set.add(value)` avec la même valeur, la méthode ne fait rien. C'est pourquoi chaque valeur est unique dans un **Set**.

Par exemple, nous souhaitons nous souvenir de tous nos visiteurs. Mais chaque visiteurs doit être unique.

Set est exactement ce qu'il nous faut :

```
let set = new Set();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// visites, certains utilisateurs viennent plusieurs fois
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set conserve une fois chaque visiteurs
alert( set.size ); // 3

for (let user of set) {
  alert(user.name); // John (puis Pete et Mary)
}
```

Set

mohamed@goumih.com

163

Nous pouvons parcourir les éléments d'un Set avec `for..of`:

ou en utilisant `forEach`:

A noter que la méthode `forEach` prend 3 arguments en paramètres:

une `value`, puis *la même valeur* `valueAgain`, et enfin le `set` lui-même.

Les méthodes pour parcourir les éléments d'une Map peuvent être utilisées :

- `set.keys()` – retourne un objet itérable contenant les valeurs,
- `set.values()` – même chose que pour `set.keys()`,
- `set.entries()` – retourne un objet itérable sous la forme de `[value, value]` pour des raisons de compatibilité avec Map

```
let set = new Set(["oranges", "apples", "bananas"]);

for (let value of set) alert(value);

// même chose en utilisant forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

```
const set1 = new Set();
set1.add(42);
set1.add('dev');

const iterator1 = set1.values();

console.log(iterator1.next().value); //42

console.log(iterator1.next().value); //dev

const set1 = new Set();
set1.add(42);
set1.add('dev');

const iterator1 = set1.entries();

for (const entry of iterator1) {
  console.log(entry);
  // Array [42, 42]
  // Array ["dev", "dev"]
}
```

EP24:

164

Utilisez Set pour stocker des valeurs uniques.

Disons que arr est un tableau.

Créez une fonction unique(arr) qui devrait renvoyer un tableau avec les éléments uniques d'arr.

Par exemple :

```
function unique(arr) {
    /* your code */
}

let values = ["Hare", "Krishna", "Hare", "Krishna",
    "Krishna", "Krishna", "Hare", "Hare", ":-0"
];

alert( unique(values) ); // Hare, Krishna, :-0
```

Anagrams sont des mots qui ont le même nombre de mêmes lettres, mais dans un ordre différent.

Par exemple :

Ecrivez une fonction aclean(arr) qui retourne un tableau nettoyé des anagrammes.

Par exemple :

```
let arr = ["nap", "teachers", "cheaters", "PAN", "ear", "era", "hectares"];
alert( aclean(arr) ); // "nap,teachers,ear" ou "PAN,cheaters,era"
```

nap - pan
ear - are - era
cheaters - hectares - teachers

WeakMap

165

La première différence entre **Map** et **WeakMap** est que les clés doivent être des objets, pas des valeurs primitives :

```
let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "ok"); // fonctionne bien (object key)

// ne peut pas utiliser une chaîne de caractères comme clé
weakMap.set("test", "Whoops"); // Erreur, parce que "test" n'est pas un objet
```

```
let john = { name: "John" };

let map = new Map();
map.set(john, "...");

john = null; // écraser la référence

// John est stocké à l'intérieur du map
// nous pouvons l'obtenir en utilisant map.keys()
```

Suppression de la mémoire avec **weakMap** :

```
let john = { name: "John" };

let weakMap = new WeakMap();
weakMap.set(john, "...");

john = null; // on écrase la référence

// John est supprimé de la mémoire !
```

```
let john = { name: "John" };

let array = [ john ];

john = null; // écraser la référence

// l'objet précédemment référencé par john est stocké dans le tableau
// donc il ne sera pas nettoyé
// nous pouvons l'obtenir sous forme de array[0]
```

WeakMap

166

WeakMap ne prend pas en charge l'itération et les méthodes **keys()**, **values()**, **entries()**, il n'y a donc aucun moyen d'en obtenir toutes les clés ou valeurs.

WeakMap n'a que les méthodes suivantes :

- **weakMap.get(key)**
- **weakMap.set(key, value)**
- **weakMap.delete(key)**
- **weakMap.has(key)**

weakMap: cas d'utilisation

167

Le principal domaine d'application de **WeakMap** est un *stockage de données supplémentaire*. Si nous travaillons avec un objet qui “appartient” à un autre code, peut-être même une bibliothèque tierce, et que nous souhaitons stocker certaines données qui lui sont associées, cela ne devrait exister que lorsque l'objet est vivant :
– alors **WeakMap** est exactement ce qu'il nous faut.
Nous plaçons les données dans un **WeakMap**, en utilisant l'objet comme clé, et lorsque l'objet est nettoyé, ces données disparaissent automatiquement également.

```
weakMap.set(john, "secret documents");
// si John meurt, les documents secrets seront détruits automatiquement
```

```
// visitsCount.js
let visitsCountMap = new WeakMap(); // map: user => visits count

// augmentons le nombre de visites
function countUser(user) {
  let count = visitsCountMap.get(user) || 0;
  visitsCountMap.set(user, count + 1);
}
```

```
// main.js
let john = { name: "John" };

countUser(john); // compter ses visites

// plus tard, John nous quitte
john = null;
```

WeakSet

168

WeakSet se comporte de la même manière :

- Il est analogue à **Set**, mais nous pouvons seulement ajouter des objets à **WeakSet** (pas de primitives).
 - Un objet existe dans le set tant qu'il est accessible ailleurs.
 - Comme Set, il prend en charge **add**, **has** et **delete**, mais pas **size**, **keys()** et aucune itération.
- Par exemple, nous pouvons ajouter des utilisateurs à WeakSet pour garder une trace de ceux qui ont visité notre site :

La limitation la plus notable de **WeakMap** et **WeakSet** est l'absence d'itérations

et l'impossibilité d'obtenir tout le contenu actuel.

Cela peut sembler gênant, mais n'empêche pas WeakMap/WeakSet de faire

leur travail principal – être un stockage “supplémentaire”

de données pour les objets qui sont stockés/gérés à un autre endroit.

```
let visitedSet = new WeakSet();

let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

visitedSet.add(john); // John nous a rendu visite
visitedSet.add(pete); // Ensuite Pete
visitedSet.add(john); // John encore

// visitedSet a 2 utilisateurs maintenant

// vérifions si John est venu
alert(visitedSet.has(john)); // true

// vérifions si Mary est venue
alert(visitedSet.has(mary)); // false

john = null;

// visitedSet sera nettoyé automatiquement
```

Object.keys, Object.values, Object.entries

169

mohamed@goumih.com

Pour les objets simples, les méthodes suivantes sont disponibles:

- **Object.keys(obj)**– retourne un tableau de clés.
- **Object.values(obj)**– retourne un tableau de valeurs.
- **Object.entries(obj)**– retourne un tableau de paires [clé, valeur].

```
1 let user = {  
2   name: "John",  
3   age: 30  
4 };
```

- `Object.keys(user) = [name, age]`
- `Object.values(user) = ["John", 30]`
- `Object.entries(user) = [["name","John"], ["age",30]]`

```
let user = {  
  name: "John",  
  age: 30  
};  
  
// boucle sur les valeurs  
for (let value of Object.values(user)) {  
  alert(value); // John, ensuite 30  
}
```

Array.From()

170

Il existe une méthode universelle qui prend une valeur itérable(objet,array-likes..) et le convertit en Array .

```
let arrayLike = {
  0: "Hello",
  1: "World",
  length: 2
};

let arr = Array.from(arrayLike); // (*)
alert(arr.pop()); // World (la méthode fonctionne)
```

```
console.log(Array.from('foo'));// résultat: Array ["f", "o", "o"]
```

```
console.log(Array.from([1, 2, 3], x => x + x));// résultat: Array [2, 4, 6]
```

Ca fonctionne avec tous les objets itérables.

```
const s = new Set(["toto", "truc", "truc", "bidule"]);
Array.from(s);
// ["toto", "truc", "bidule"]

// Map
const m = new Map([[1, 2], [2, 4], [4, 8]]);
Array.from(m);
// [[1, 2], [2, 4], [4, 8]]
```

Object.fromEntries()

Les objets manquent de nombreuses méthodes existantes pour les tableaux, par exemple map, filter .. Si nous souhaitons leur appliquer ces méthodes, nous pouvons utiliser Object.entries suivis par Object.fromEntries :

- 1.Utilisons Object.entries(obj) pour obtenir un tableau de paires clé / valeur de obj.
- 2.Utilisons des méthodes de tableau sur ce tableau, par exemple map, pour transformer ces paires clé / valeur.
- 3.Utilisons Object.fromEntries(array) sur le tableau résultant pour le reconvertisr en objet.

Par exemple, nous avons un objet avec des prix et aimerais les doubler :

```
let prices = {  
    banana: 1,  
    orange: 2,  
    meat: 4,  
};  
  
let doublePrices = Object.fromEntries(  
    // convertir les prix en tableau, mapper chaque paire clé / valeur dans  
    // puis fromEntries restitue l'objet  
    Object.entries(prices).map(entry => [entry[0], entry[1] * 2])  
);  
  
alert(doublePrices.meat); // 8
```

Objets et récursivité

172

nous voulons une fonction pour obtenir la somme de tous les salaires d'un objet company imbriqué.
Comment peut-on faire ça?

```
let company = { // le même objet, compressé pour la brièveté
    sales: [{name: 'John', salary: 1000}, {name: 'Alice', salary: 1600 }],
    development: {
        sites: [{name: 'Peter', salary: 2000}, {name: 'Alex', salary: 1800 }],
        internals: [{name: 'Jack', salary: 1300}]
    }
};

// La fonction pour faire le travail
function sumSalaries(department) {
    if (Array.isArray(department)) { // case (1)
        return department.reduce((prev, current) => prev + current.salary, 0); // addition
    } else { // case (2)
        let sum = 0;
        for (let subdep of Object.values(department)) {
            sum += sumSalaries(subdep); // appel récursivement pour les sous-département
        }
        return sum;
    }
}

alert(sumSalaries(company)); // 7700
```

Il existe un objet **salaries** avec un nombre arbitraire de salaires.
Ecrivez la fonction **sumSalaries(salaries)** qui retourne la somme des salaires en utilisant **Object.values** et la boucle **for..of**.

Si salaries est vide, le résultat doit être 0.
Par exemple:

```
let salaries = {  
    "John": 100,  
    "Pete": 300,  
    "Mary": 250  
};
```

```
alert( sumSalaries(salaries) ); // 650
```

Créer la fonction **topSalary(salaries)** qui renvoie le nom de la personne la mieux payée.

- Si salaries est vide, devrait retourner null.
- S'il y a plusieurs personnes les mieux payées, renvoyez-en une.

P.S. Utilisez **Object.entries**



Ecrivez la fonction **count(obj)** qui retourne le nombre de propriétés qu'il y a dans l'objet:

```
let user = {  
    name: 'John',  
    age: 30  
};
```

```
alert( count(user) ); // 2
```

EP26:

174

Créer un **weakSet ReadMessages** qui permet de stocker les éléments de **messages** sans doublons
 vérifier que **weakSet** contient ses deux éléments
 ,supprimer le premier élément de **messages** et vérifier que l'élément a été supprimé de **ReadMeassages**.

Créer **weakMap readMap** pour stocker le premier message Avec Date de création.

```
let messages = [
  {text: "Hello", from: "John"}, 
  {text: "How goes?", from: "John"}, 
  {text: "How goes?", from: "John"}]
```

Créez deux fonctions **setData** et **getData** qui renvoie données associées à un objet donné à l'aide d'un **WeakMap** et ses méthodes(set et get).

Exemple :

```
const obj = {};
setData(obj, { name: 'John', age: 30 });
console.log(getData(obj)); // { name: 'John', age: 30 }
```

Créez une fonction **deleteObject()** qui prend un objet et supprime de manière récursive toutes ses propriétés et ses objets enfants. Utilisez un WeakSet pour garder une trace des objets qui ont déjà été supprimés pour éviter les boucles infinies.

Exemple :

```
const obj = { name: 'John', address: { city: 'New York', country: 'USA' } };
deleteObject(obj);
console.log(obj); // {}
```

L'affectation par décomposition

175

L'affectation par décomposition est une syntaxe spéciale qui nous permet de “décompresser” des tableaux ou des objets dans un ensemble de variables.

La décomposition fonctionne également très bien avec des fonctions complexes comportant de nombreux paramètres, valeurs par défaut, etc.

Décomposition d'un tableau

```
// nous avons un tableau avec le prénom et le nom
let arr = ["John", "Smith"]
```

```
// l'affectation par décomposition
// sets firstName = arr[0]
// and surname = arr[1]
let [firstName, surname] = arr;

alert(firstName); // John
alert(surname); // Smith
```

```
let [firstName, surname] = "John Smith".split(' ');
alert(firstName); // John
alert(surname); // Smith
```

```
// let [firstName, surname] = arr;
let firstName = arr[0];
let surname = arr[1];
```

Les éléments indésirables du tableau peuvent également être supprimés via une virgule supplémentaire :

```
// second element is not needed
let [firstName, , title] = ["Julius", "Caesar", "Consul",
                           alert( title ); // Consul
```

En fait, nous pouvons l'utiliser avec n'importe quel itérable, pas seulement les tableaux :

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

Avec objet aussi:

```
let user = {};
[user.name, user.surname] = "John Smith".split(' ');

alert(user.name); // John
alert(user.surname); // Smith
```

Opérateur Spread:

```
let [item1 = default, item2, ...rest] = array
let {prop : varName = default, ...rest} = object
```

L'affectation par décomposition

176

nous avons vu la méthode `Object.entries(obj)`

Nous pouvons l'utiliser avec la décomposition pour boucler sur les clés et valeurs d'un objet :

```
let user = {
  name: "John",
  age: 30
};

// boucler sur les clés et les valeurs
for (let [key, value] of Object.entries(user)) {
  alert(` ${key}: ${value}`); // name: John, ensuite age: 30
}
```

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");

// Map iterates as [key, value] pairs, very convenient for destructuring
for (let [key, value] of user) {
  alert(` ${key}: ${value}`); // name: John, ensuite age: 30
}
```

Si un objet ou un tableau contient d'autres objets et tableaux imbriqués, nous pouvons utiliser des modèles à gauche plus complexes pour extraire des parties plus profondes.

```
let {
  size: {
    width,
    height
  },
  items: [item1, item2],
  title = "Menu"
}
```

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
}
```

```
let options = {
  size: {
    width: 100,
    height: 200
  },
  items: ["Cake", "Donut"],
  extra: true
};

let options = {
  size: { // mettre la taille ici
    width,
    height
  },
  items: [item1, item2], // attribuer des éléments ici
  title = "Menu"
} = options;
```

Affectation par décomposition

177

Exemple avec les fonctions :

```
function sum(x, y, z) {  
    return x + y + z;  
}  
  
const numbers = [1, 2, 3];  
  
console.log(sum(...numbers)); // 6
```

Exemple avec les objets:

```
var profil = { prenom: 'Sarah', profilComplet: false };  
var profilMisAJour = { nom: 'Dupont', profilComplet: true };  
  
var clone = { ...profil };  
// Object { prenom: 'Sarah', profilComplet: false }  
  
var fusion = { ...profil, ...profilMisAJour };  
// Object { prenom: 'Sarah', nom: 'Dupont', profilComplet: true };
```

Exemple avec les tableaux :

```
var articulations = ['épaules', 'genoux'];  
var corps = ['têtes', ...articulations, 'bras', 'pieds'];  
// ["têtes", "épaules", "genoux", "bras", "pieds"]
```

```
var arr = [1, 2, 3];  
var arr2 = [...arr];  
arr2.push(4);
```

```
console.log(arr2); // [1, 2, 3, 4]  
console.log(arr); // [1, 2, 3] (inchangé)
```

```
var arr1 = [0, 1, 2];  
var arr2 = [3, 4, 5];  
arr1 = [...arr1, ...arr2]; // arr1 vaut [0, 1, 2, 3, 4, 5]
```

EP27:

178

Écrivez l'affectation par décomposition qui se lit comme suit :

- La propriété name dans la variable name.
- La propriété years dans la variable age.
- La propriété isAdmin dans la variable isAdmin
(false si absent)

```
let user = {  
    name: "John",  
    years: 30  
};
```

```
let user = { name: "John", years: 30 };  
  
// votre code à gauche ::  
// ... = user  
  
alert( name ); // John  
alert( age ); // 30  
alert( isAdmin ); // false
```

Déstructure Le tableau users en 3 d'objets:
User1, user2, user3

```
let users =[ {name:"user1",age:34},  
{name:"user2",age:25},{name:"user3",age:20},  
 ]
```

Créer une fonction qui affiche les données
d'une personne et tester la sur l'objet
person

```
function identité({name, age, address})
```

```
let person = {  
    name: 'person1',  
    age: 30,  
    address: '123 St,MA'  
};
```

1. soit l'objet suivant:

```
const objet = { a: 1, b: 2, c: 3 };
```

.Affecter les valeurs des attributs de objet a trois variables a,b,c en utilisant la déconstruction, afficher les valeurs.

2. Même chose avec l'objet :
`const objet2 = { a1: 1, b1: { c1: 2, d1: 3 } };`

3. Combiner objet et object2 dans objet3

Soit la chaîne suivante :
`const filiere = "DEV DIGITAL FULL-STACK MOBILE";`

Utiliser la décomposition pour affecter le premier mot au variable dev1, le 2eme à dev2, le reste au spécialité.

Soit l'objet suivant :

```
const fruits = [
  { nom: "pomme", couleur: "rouge", prix: 1.00 },
  { nom: "banane", couleur: "jaune", prix: 0.50 },
  { nom: "orange", couleur: "orange", prix: 0.75 }
];
```

créer une nouvelle liste de fruits, en ajoutant un quatrième fruit **kiwi** à la liste d'origine. Cependant, vous ne voulez pas modifier la liste d'origine

Partie 7 : Stockage interne

- Les types de stockages
- Les méthodes des différents types
 - Exemples
 - localStorage
 - sessionStorage

LocalStorage vs sessionStorage vs cookies

181

	cookies	Local Storage	Session Storage
Capacité	4kb	10mb	5mb
Navigateurs	HTML4/HTML5	HTML5	HTML5
Expiration	configuré manuellement	jamais	Après la fermeture du l'onglet
L'endroit du stockage	Navigateur et serveur	Navigateur	Navigateur
Envoi avec requêtes	oui	non	non
Accessibilité	N'importe quelle fenêtre	N'importe quelle fenêtre	L'onglet du travail

Méthodes de local et session

182

Les objets de stockage Web **localStorage** et **sessionStorage** permettent d'enregistrer les paires clé/valeur dans le navigateur.

Ce qui est intéressant à leur sujet, c'est que les données survivent à une actualisation de la page (pour **sessionStorage**) et même à un redémarrage complet du navigateur (pour **localStorage**)

La plupart des navigateurs autorisent au moins 5 mégaoctets de données (ou plus) et ont des paramètres pour configurer cela.

Les deux objets de stockage fournissent les mêmes méthodes et propriétés :

- `setItem(key, value)` – stocke la paire clé/valeur.
- `getItem(key)` – récupère la valeur par clé.
- `removeItem(key)` – supprime la clé avec sa valeur.
- `clear()` – supprime tout.
- `key(index)` – récupère la clé sur une position donnée.
- `length` – le nombre d'éléments stockés.

LocalStorage vs sessionStorage vs cookies

183

Voici des exemples :

```
<script>
localStorage.setItem('lunch', 'cereal');
console.log(localStorage.getItem('breakfast'));

//pour supprimer une donnée
localStorage.removeItem('lunch');
//pour supprimer toutes le données
localStorage.clear();

//localStorage c'est comme sessionStorage

//pour manipulé les cookies
document.cookie = "hello=true";
document.cookie = "doSomethingOnlyOnce=true; expires=Fri, 31 Dec 9999 23:59:59 GMT";
document.cookie = "person=beau; expires=Fri, 31 Dec 9999 23:59:59 GMT; path=/"
document.cookie = "person=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;";
console.log(document.cookie)
</script>
```

localStorage

mohamed@goumih.com

184

Les principales caractéristiques de **localStorage** sont les suivantes :

- Partagé entre tous les onglets et fenêtres d'une même origine.
- Les données n'expirent pas. Il reste après le redémarrage du navigateur et même le redémarrage du système d'exploitation.

```
const array=["Adam","Eve"];
//convertir array to string
let names=JSON.stringify(array);
localStorage.setItem("names",names);
console.log("avant",names);

names=localStorage.getItem("names");
//convertir string to array
names=JSON.parse(names);
console.log("après",names);
```

```
//utiliser des boucles pour boucler les
éléments
for (let i=0; i<localStorage.length; i++) {
    let key = localStorage.key(i);
    console.log(` ${key}: ${localStorage.getItem(key)} `);
}
for(let key in localStorage) {
    if (!localStorage.hasOwnProperty(key))
        //filtrer les champs
        // vérification hasOwnProperty :
        continue; // sauter des clés comme
"setItem", "getItem" etc.
    }
    console.log(` ${key}: ${localStorage.getItem(key)} `);
}
```

Session Storage

L'objet **sessionStorage** est beaucoup moins utilisé que **localStorage**.

Les propriétés et les méthodes sont les mêmes, mais c'est beaucoup plus limité :

- Le **sessionStorage** n'existe que dans l'onglet actuel du navigateur.
 - Un autre onglet avec la même page aura un stockage différent.
 - Mais il est partagé entre les iframes du même onglet (en supposant qu'ils proviennent de la même origine).
- Les données survivent à l'actualisation de la page, mais pas à la fermeture/ouverture de l'onglet.

```
sessionStorage.setItem('test', 1);
  // actualisez la page vous pouvez toujours obtenir les données :
  console.log( sessionStorage.getItem('test') ); // après
rafraîchissement : 1

  //...Mais si vous ouvrez la même page dans un autre onglet et
réessayez,
  // le code ci-dessus renvoie null, ce qui signifie "rien trouvé".
```

EP29:

186

Créez une page Web avec un formulaire qui permet à l'utilisateur d'entrer une chaîne et de la stocker dans localStorage. Lorsque la page est chargée, récupérez la chaîne de localStorage et affichez-la sur la page.

Insérer une chaîne :

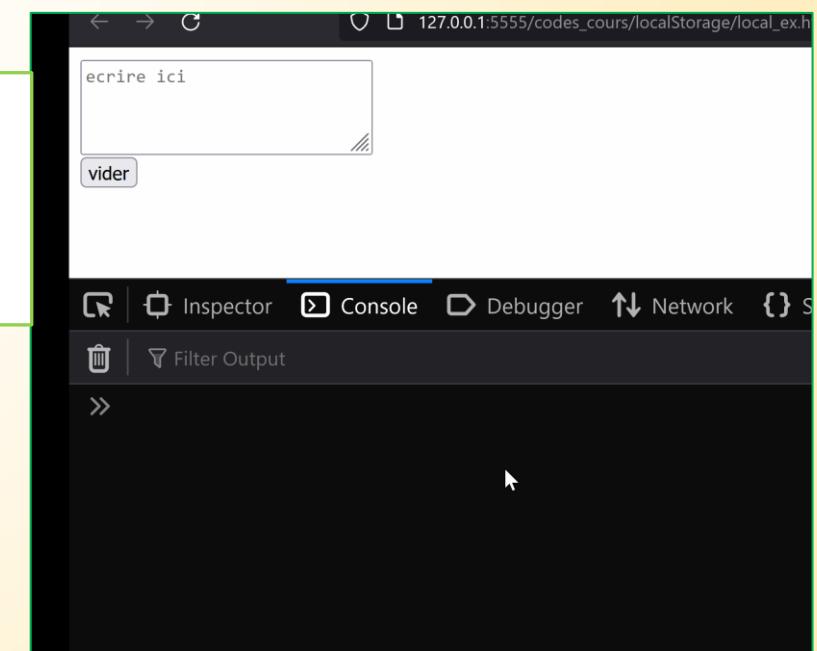
javascript

Créez une fonction qui prend un objet, le sérialise en JSON et le stocke dans localStorage.

Créez une autre fonction qui récupère la chaîne JSON de localStorage, la déserialise en un objet et renvoie l'objet.

Créez un textarea qui permet de taper et enregistrer en même temps dans le storage ,lorsque on clique le button **vider** il supprime la donnée du stockage.

Créez un bouton lorsque on clique il efface toutes les données stockées dans localStorage.



Partie 8:

Manipulation du

DOM

avec Javascript

Browser: Document, Events, Interfaces

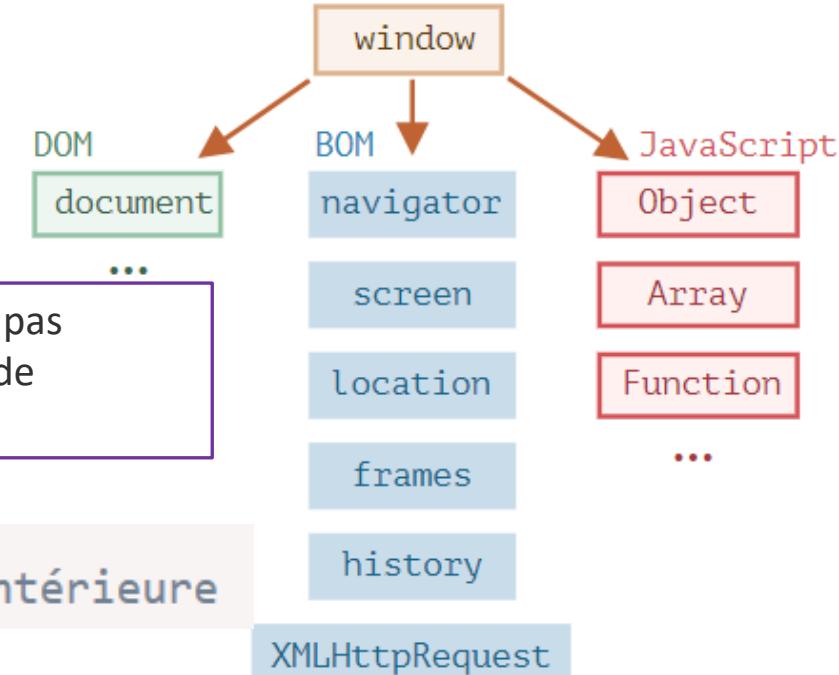
188

Voici une vue globale de ce que nous avons lorsque JavaScript s'exécute dans un navigateur Web :

```
// les fonctions globales sont des méthodes de l'objet global :  
window.sayHi();
```

Les fonctions `alert`/`confirm`/`prompt` font aussi partie du BOM : elles ne sont pas directement liées au document, mais représentent des méthodes du navigateur de communication pure avec l'utilisateur.

```
alert(window.innerHeight); // hauteur de la fenêtre intérieure
```



le BOM fait partie de la [spécification HTML](#) générale.

Oui, vous avez bien entendu. La spécification HTML disponible à l'adresse <https://html.spec.whatwg.org>

Il existe également une spécification distincte, [Modèle d'objet CSS \(CSSOM\)](#) pour les règles CSS et les feuilles de style, qui explique comment elles sont représentées en tant qu'objets et comment les lire et les écrire.

Document Object Model, ou DOM en abrégé, représente tout le contenu de la page sous forme d'objets pouvant être modifiés. L'objet `document` est le “point d’entrée” principal de la page. Nous pouvons changer ou créer n’importe quoi sur la page en l’utilisant.

DOM

189

```
alert(location.href); // affiche l'URL actuelle  
if (confirm("Go to Wikipedia?")) {  
    location.href = "https://wikipedia.org"; // rediriger le navigateur vers une autre  
}
```

Le modèle d'objet du navigateur (BOM en anglais) contient des objets supplémentaires fournis par le navigateur (l'environnement hôte) pour travailler avec tout à l'exception du document.

Par exemple :

- L'objet navigator fournit des informations contextuelles à propos du navigateur et du système d'exploitation.
- Il y a beaucoup de propriétés mais les deux plus connues sont : navigator.userAgent – qui donne des informations sur le navigateur actuel,
- et navigator.platform sur la plateforme (peut permettre de faire la différence entre Windows/Linux/Mac etc).
- L'objet location nous permet de lire l'URL courante et peut rediriger le navigateur vers une nouvelle adresse.

```
// change la couleur de fond en rouge  
document.body.style.background = "red";  
  
// réinitialisation après 1 seconde  
setTimeout(() => document.body.style.background = "", 1000);
```

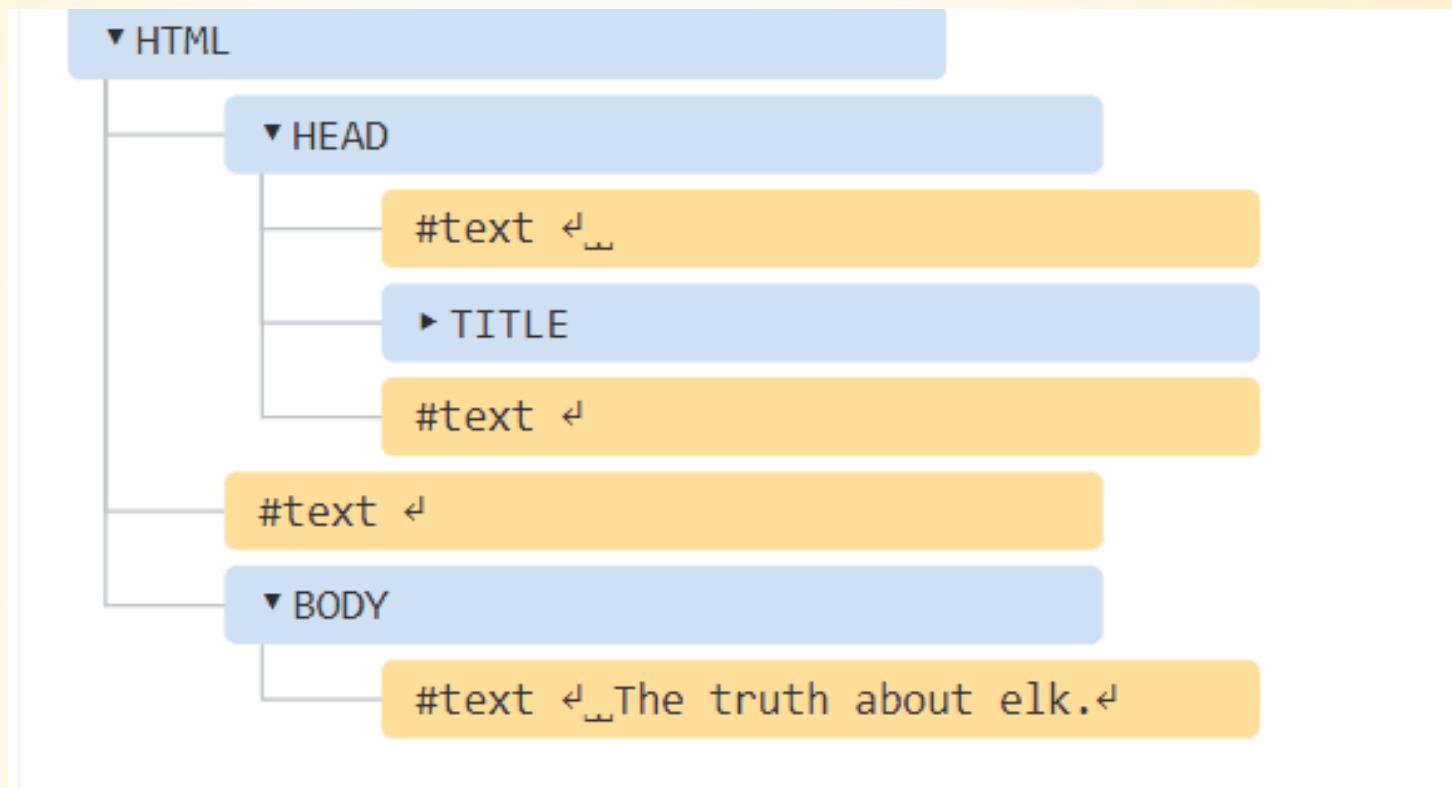
Ici on a utilisé `document.body.style`, mais il y a bien plus encore. Les propriétés et les méthodes sont décrites dans la spécification : [DOM Living Standard](#).

Un exemple du DOM

mohamed@goumih.com

190

Le DOM représente le HTML comme une structure arborescente de balises. Voici à quoi ça ressemble :



Les balises sont des *nœuds d'élément* (ou simplement des éléments) et forment la structure arborescente : `<html>` est à la racine, puis `<head>` et `<body>` sont ses enfants, etc.

Pour voir la structure dom en temps réel, essayez le [live dom viewer](#). Tapez simplement le document, et il apparaîtra comme un dom en un instant.

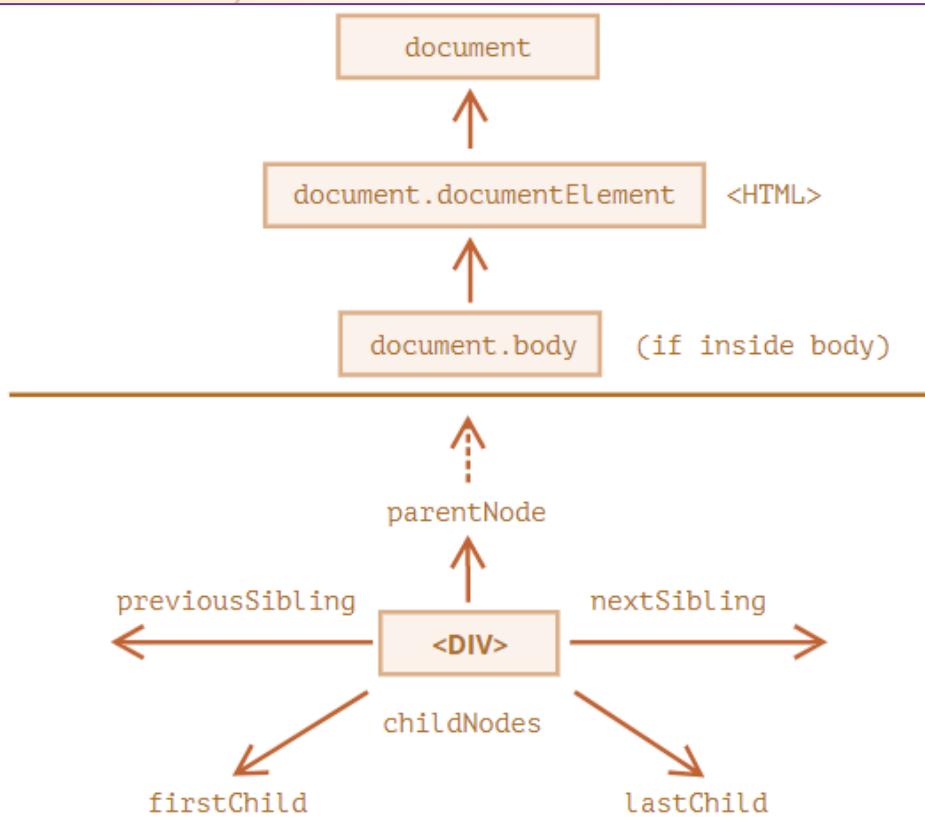
Parcourir le DOM

191

Le DOM nous permet de faire n'importe quoi avec les éléments et leur contenu, mais nous devons d'abord atteindre l'objet DOM correspondant.

Toutes les opérations sur le DOM commencent par l'objet `document`. C'est le “point d'entrée” principal du DOM. De là, nous pouvons accéder à n'importe quel nœud.

Voici une image des liens qui permettent de voyager entre les nœuds DOM :



si un script se trouve dans `<head>`, alors `document.body` n'est pas disponible, car le navigateur ne l'a pas encore lu.
Ainsi, dans l'exemple ci-dessous, la première alert affiche null :

```

<head>
  <script>
    alert( "From HEAD: " + document.body ); // null, il n'y a pas encore d
  </script>
</head>

<body>

  <script>
    alert( "From BODY: " + document.body ); // HTMLBodyElement maintenant
  </script>

</body>
  
```

Enfants : childNodes, firstChild, lastChild

192

mohamed@goumih.com

Nous utiliserons désormais deux termes :

- **Noeuds enfants (ou enfants)** – éléments qui sont des enfants directs. En d'autres termes, ils sont imbriqués dans celui donné. Par exemple, `<head>` et `<body>` sont des enfants de l'élément `<html>`.

- **Descendants** – tous les éléments imbriqués dans l'élément donné, y compris les enfants, leurs enfants, etc.

Par exemple, `<body>` a des enfants `<div>` et `` (et quelques nœuds texte vides) :

Les propriétés `firstChild` et `lastChild` donnent un accès rapide aux premier et dernier enfants. La collection `childNodes` répertorie tous les nœuds enfants, y compris les nœuds texte

`elem.childNodes[0] === elem.firstChild`

`elem.childNodes[elem.childNodes.length - 1] === elem.lastChild`

Il y a aussi une fonction spéciale `elem.hasChildNodes()` pour vérifier s'il y a des nœuds enfants.

L'exemple ci-dessous montre des enfants de `document.body` :

```
<body>
  <div>Begin</div>

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

  <script>
    for (let i = 0; i < document.body.childNodes.length; i++) {
      alert( document.body.childNodes[i] ); // Text, DIV, Text, UL, ..., SCRIPT
    }
  </script>
  ...more stuff...
</body>
```

Collection DOM

193

Les frères et sœurs sont des nœuds qui sont les enfants du même parent.
Par exemple, <head> et <body> sont des frères et sœurs :

<body> est dit le frère “suivant” ou “droit” de <head>,
•<head> est dit le frère “précédent” ou “gauche” de <body>. Le frère suivant est dans la propriété nextSibling, et le précédent dans previousSibling. Le parent est disponible en tant que parentNode.

```
// le parent de <body> est <html>
alert( document.body.parentNode === document.documentElement ); // true

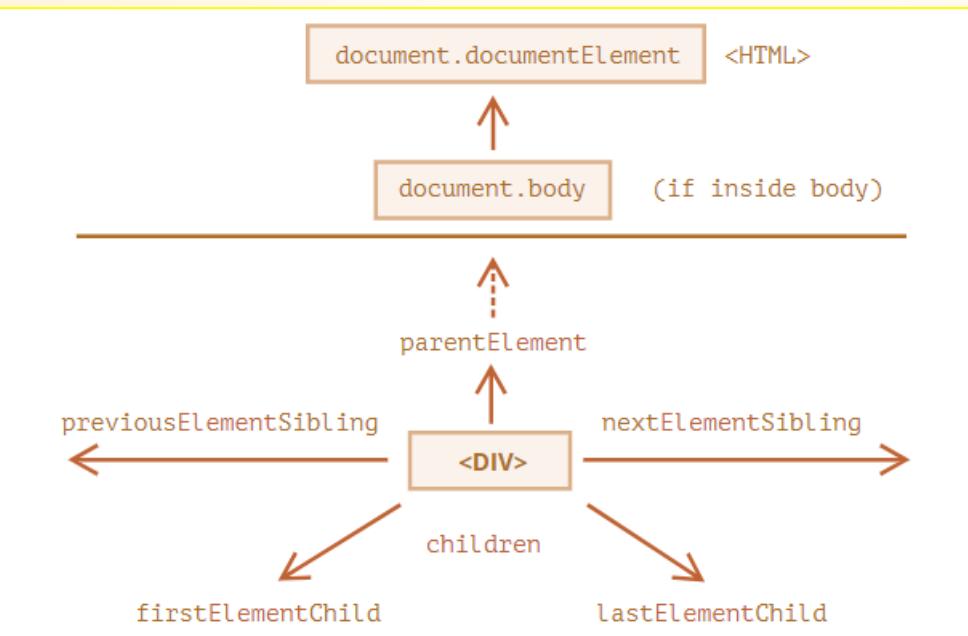
// après <head> vient <body>
alert( document.head.nextSibling ); // HTMLBodyElement

// avant <body> vient <head>
alert( document.body.previousSibling ); // HTMLHeadElement
```

Navigation par élément uniquement

194

Voyons donc plus de liens de navigation qui ne prennent en compte que les *nœuds élément* :



```

while(elem = elem.parentElement) { // remonter jusqu'à <html>
  alert( elem );
}
  
```

```

<body>
  <div>Begin</div>

  <ul>
    <li>Information</li>
  </ul>

  <div>End</div>

  <script>
    for (let elem of document.body.children) {
      alert(elem); // DIV, UL, DIV, SCRIPT
    }
  </script>
  
```

Parcourir les tableaux

mohamed@goumih.com

195

L'élément **<table>** supporte ces propriétés :

- **table.rows** – la collection d'éléments **<tr>** du tableau.
- **table.caption/tHead/tFoot** – références aux éléments **<caption>**, **<thead>**, **<tfoot>**.
- **table.tBodies** – la collection d'éléments **<tbody>** (peut être multiple selon la norme, mais il y en aura toujours au moins une – même s'elle n'est pas dans le HTML source, le navigateur la mettra dans le DOM).

<thead>, <tfoot>, <tbody> les éléments fournissent la propriété **rows** :

- **tbody.rows** – la collection de **<tr>** à l'intérieur.

<tr>:

- **tr.cells** – la collection de cellules **<td>** et **<th>** à l'intérieur du **<tr>** donné.
- **tr.sectionRowIndex** – la position (index) du **<tr>** donné à l'intérieur du **<thead>/<tbody>/<tfoot>**.
- **tr.rowIndex** – le nombre de **<tr>** dans le tableau dans son ensemble (y compris toutes les lignes du tableau).

****<td>** et **<th>** : **

- **td.cellIndex** – le numéro de la cellule à l'intérieur du **<tr>** qui l'entoure.

```
<table id="table">
  <tr>
    <td>one</td><td>two</td>
  </tr>
  <tr>
    <td>three</td><td>four</td>
  </tr>
</table>

<script>
  // obtenir td avec "two" (première ligne, deuxième colonne)
  let td = table.rows[0].cells[1];
  td.style.backgroundColor = "red"; // le mettre en valeur
</script>
```

document.getElementById

196

- Si un élément a l'attribut **id**, nous pouvons obtenir l'élément en utilisant la méthode **document.getElementById(id)**, peu importe où il se trouve.
- **L'identifiant doit être unique** : Il ne peut y avoir qu'un seul élément dans le document avec l'identifiant donné. S'il y a plusieurs éléments avec le même identifiant, alors le comportement des méthodes qui l'utilisent est imprévisible, par ex. **document.getElementById** peut renvoyer n'importe lequel de ces éléments au hasard. Veuillez donc respecter la règle et garder un identifiant unique.

```
<div id="elem">
  <div id="elem-content">Element</div>
</div>

<script>
  // get the element
  let elem = document.getElementById('elem');

  // make its background red
  elem.style.background = 'red';
</script>
```

querySelectorAll/querySelector

197

- la méthode la plus polyvalente,**document.querySelectorAll(css)** renvoie tous les éléments à l'intérieur de document correspondant au sélecteur CSS donné.
- Cette méthode est en effet puissante, car n'importe quel sélecteur est sélectionné.
- Les pseudo-classes dans le sélecteur CSS telles que :hover et :active sont également prises en charge:
Par exemple, **document.querySelectorAll(':hover')** renverra la collection avec les éléments sur lesquels le pointeur se trouve maintenant

Ici, nous recherchons tous les éléments qui sont les derniers enfants :

L'appel à **document.querySelector(css)** renvoie le premier élément pour le sélecteur CSS donné. En d'autres termes, le résultat est le même que **document.querySelectorAll(css)[0]**. C'est donc plus rapide et aussi plus court à.

```
<ul>
  <li>The</li>
  <li>test</li>
</ul>
<ul>
  <li>has</li>
  <li>passed</li>
</ul>
<script>
  let elements = document.querySelectorAll('ul > li:last-child');

  for (let elem of elements) {
    alert(elem.innerHTML); // "test", "passed"
  }
</script>
```

matches/closest

mohamed@goumih.com

198

elem.matches(css) : vérifie si elem correspond au sélecteur CSS donné. Il renvoie vrai ou faux. La méthode est pratique lorsque nous parcourons des éléments (comme dans un tableau ou quelque chose du genre).

```
<a href="http://example.com/file.zip">...</a>
<a href="http://ya.ru">...</a>

<script>
    ...
    for (let elem of document.body.children) {
        if (elem.matches('a[href$="zip"]')) {
            alert("The archive           : " + elem.href );
        }
    }
</script>
```

Les ancêtres d'un élément sont : parent, le parent de parent, son parent et ainsi de suite.

La méthode **elem.closest(css)** recherche l'ancêtre le plus proche qui correspond au sélecteur CSS. L'élément lui-même est également inclus dans la recherche

```
<h1>Contents</h1>

<div class="contents">
    <ul class="book">
        <li class="chapter">Chapter 1</li>
        <li class="chapter">Chapter 1</li>
    </ul>
</div>
```

```
<script>
    let chapter = document.querySelector('.chapter'); // LI

    alert(chapter.closest('.book')); // UL
    alert(chapter.closest('.contents')); // DIV

    alert(chapter.closest('h1')); // null (because h1 is not an ancestor)
</script>
```

getElementsBy*

199

document.getElementsByTagName(tag) : recherche les éléments avec la balise donnée et en renvoie la collection.
Le paramètre de balise peut également être une étoile "*" pour "toutes les balises".

document.getElementsByClassName(className): renvoie les éléments qui ont la **classe** CSS donnée.

document.getElementsByName(name): renvoie les éléments avec l'attribut **name** donné, à l'échelle du document.
(Très rarement utilisé).

Toutes les méthodes "**getElementsBy***" renvoient une collection dynamique. Ces collections reflètent toujours l'état actuel du document et se "mettent à jour automatiquement" lorsqu'il change.

```
<form name="my-form">
  <div class="article">Article</div>
  <div class="long article">Long article</div>
</form>

<script>
  // find by name attribute
  let form = document.getElementsByName('my-form')[0];

  // find by class inside the form
  let articles = form.getElementsByClassName('article');
  alert(articles.length); // 2, found two elements with class "article"
</script>
```

```
let divs = document.getElementsByTagName('div');
alert(divs.length); // 1
```

```
let divs = document.getElementsByTagName('div');
```

EP30:

200

Accéder ou élément div li ,ul en utilisant DOM

```
<html>
<body>
<div>DD:</div>
<ul>
<li>Dev101</li>
<li>Dev102</li>
</ul>
</body> </html>
```

Réaliser le tableau suivant en utilisant html , et colorer les diagonales en utilisant DOM

1:1	2:1	3:1	4:1	5:1
1:2	2:2	3:2	4:2	5:2
1:3	2:3	3:3	4:3	5:3
1:4	2:4	3:4	4:4	5:4
1:5	2:5	3:5	4:5	5:5

- ▶ Sélectionner tous les inputs et déterminer leur nombres et les afficher et leurs valeurs

```
<table id="table">
  <tr>
    <td>Your age:</td>

    <td>
      <label>
        <input type="radio" name="age" value="young" checked> less than 18
      </label>
      <label>
        <input type="radio" name="age" value="mature"> from 18 to 50
      </label>
      <label>
        <input type="radio" name="age" value="senior"> more than 60
      </label>
    </td>
  </tr>
</table>
```

Lorem ipsum dolor sit amet, consectetur adipisicing elit, enim ad minim veniam, quis nostrud exercitation ullamco

Changer Style

- ▶ Créer une page html contient une paragraphe de id=p1 et un button
- ▶ Créer une fonction **changeStyle()** qui change la couleur et background et border lorsqu 'on clique sur le button **Changer style**

Lorem ipsum dolor sit amet, consectetur adipisicing elit, enim ad minim veniam, quis nostrud exercitation ullamco

Changer Style

Dans un fichier HTML, ajouter des alertes :

- 1.au chargement de la page,**
- 2.quand on passe sur une image,**
- 3.quand on clique sur un bouton,**

Ajouter un bouton qui, quand on clique dessus, fait appel à document.write() pour Avec des boutons :

- 1.changer la couleur de fond du document quand on clique sur un bouton,**
- 2.changer aussi l'apparence du bouton,**
- 3.ajouter un bouton qui permette de revenir à la normale.**

Sur des images :

- 1.changer une image par une autre quand on clique dessus,**
- 2.remettre l'image d'origine quand on clique à nouveau**
- 3.Écrire un script qui permet de créer, au chargement de la page, une nouvelle fenêtre (window.open()) qui se fermera automatiquement (window.close()) au bout de 3 secondes.**

Classes de nœuds DOM

203

Différents nœuds DOM peuvent avoir des propriétés différentes.

Par exemple, un nœud élément correspondant à la balise `<a>` a des propriétés liées aux liens, et celui correspondant à `<input>` a des propriétés liées aux entrées, etc. Les nœuds texte ne sont pas identiques aux nœuds élément.

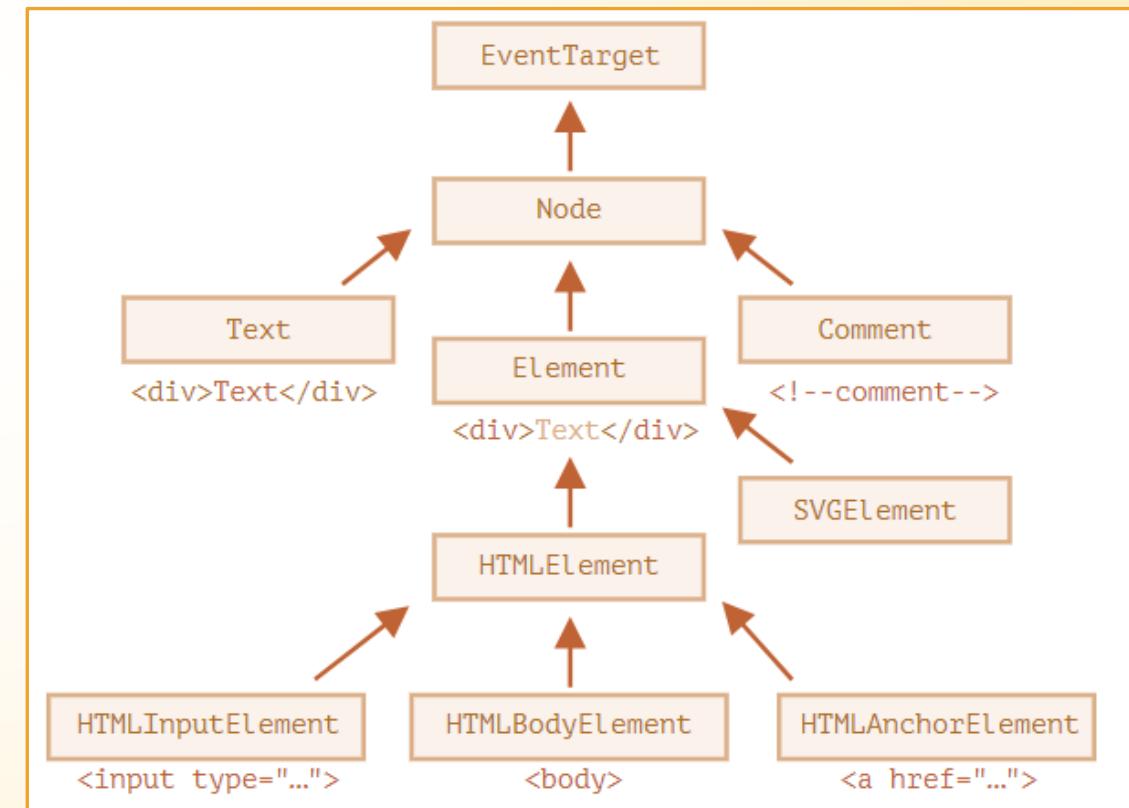
Mais il existe également des propriétés et des méthodes communes à chacun d'entre eux, car toutes les classes de nœuds DOM forment une hiérarchie unique.

Chaque nœud DOM appartient à la classe intégrée correspondante.

Étant donné un nœud DOM, nous pouvons lire son nom de balise dans les propriétés `nodeName` ou `tagName`

```
alert( document.body.nodeName ); // BODY  
alert( document.body.tagName ); // BODY
```

```
// pour le commentaire  
alert( document.body.firstChild.tagName ); // undefined (pas un élément)  
alert( document.body.firstChild.nodeName ); // #comment
```



innerHTML: les contenus

204

La propriété **innerHTML** permet d'obtenir le HTML à l'intérieur de l'élément sous forme de chaîne de caractères. Nous pouvons également le modifier. C'est donc l'un des moyens les plus puissants de modifier la page. L'exemple montre le contenu de **document.body** puis le remplace complètement :

```
<body>
  <p>A paragraph</p>
  <div>A div</div>

  <script>
    alert( document.body.innerHTML ); // lit le contenu actuel
    document.body.innerHTML = 'The new BODY!';
    // le remplace
  </script>

</body>
```

Nous pouvons ajouter du HTML à un élément en utilisant **elem.innerHTML+= ""**

```
Div.innerHTML += "<div>Hello<img
src='smile.gif' /> !</div>";
Div.innerHTML += « World»;
```

La propriété **outerHTML** contient le code HTML complet de l'élément. C'est comme **innerHTML** plus l'élément lui-même. **outerHTML ne modifie pas l'élément. Au lieu de cela, il le remplace dans le DOM**

```
<div>Hello, world!</div>

<script>
  let div = document.querySelector('div');

  // remplace div.outerHTML avec <p>...</p>
  div.outerHTML = '<p>A new element</p>'; // (*)

  // Wow! 'div' est toujours la même !
  alert(div.outerHTML); // <div>Hello, world!</div> (**)
</script>
```

nodeValue , Data, textContent ,hidden

205

les propriétés **nodeValue** et **data**: permet de retourner le contenu.

Ces deux sont presque les mêmes pour une utilisation pratique,
il n'y a que des différences de spécifications mineures.

Nous allons donc utiliser **data**, car il est plus court.

Un exemple de lecture du contenu d'un nœud texte et d'un commentaire

```
<body>
  Hello
  <!-- Commentaire -->
  <script>
    let text = document.body.firstChild;
    alert(text.data); // Hello

    let comment = text.nextSibling;
    alert(comment.data); // Commentaire
  </script>
</body>
```

textContent: donne accès au *texte* à l'intérieur de l'élément : seulement le texte, moins tous les **<tags>**.

```
<div id="elem1"></div>
<div id="elem2"></div>

<script>
  let name = prompt("What's your name?", "<b>Winnie-the-Pooh!</b>");

  elem1.innerHTML = name;
  elem2.textContent = name;
</script>
```

```
<div id="news">
  <h1>Headline!</h1>
  <p>Martians attack people!</p>
</div>

<script>
  // Headline! Martians attack people!
  alert(news.textContent);
</script>
```

L'attribut “**hidden**” : et la propriété DOM spécifient si l'élément est visible ou non.

Nous pouvons l'utiliser dans le HTML ou l'attribuer en utilisant JavaScript, comme ceci :

```
<div>Both divs below are hidden</div>

<div hidden>With the attribute "hidden"</div>

<div id="elem">JavaScript assigned the property "hidden"</div>

<script>
  elem.hidden = true;
</script>
```

Autres propriétés

206

Les éléments DOM ont également des propriétés supplémentaires, en particulier celles qui dépendent de la classe :

- **value** : la valeur
- Type pour `<input>`, `<select>` et `<textarea>`..
- **Href** :pour ``
- **id** : la valeur de l'attribut "id", pour tous les éléments
- ...et beaucoup plus...

```
<input type="text" id="elem" value="value">

<script>
  alert(elem.type); // "text"
  alert(elem.id); // "elem"
  alert(elem.value); // value
</script>
```

Si nous voulons connaître la liste complète des propriétés prises en charge pour une classe donnée, nous pouvons les trouver dans la spécification.

Par exemple, `HTMLInputElement` est documenté à <https://html.spec.whatwg.org/#htmlinputelement>.

Ou si nous voulons les obtenir rapidement ou encore si nous sommes intéressés par une spécification concrète de navigateur

– nous pouvons toujours sortir l'élément en utilisant `console.dir(elem)` et lire les propriétés. Ou explorez les "propriétés DOM" dans l'onglet Éléments des outils de développement du navigateur.

Propriétés DOM

207

Les nœuds DOM sont des objets JavaScript normaux.

Nous pouvons les modifier.

Par exemple, créons une nouvelle propriété dans **document.body** :

```
document.body.myData = {  
    name: 'Caesar',  
    title: 'Imperator'  
};
```

```
alert(document.body.myData.title); // Imperator
```

Nous pouvons également ajouter une méthode :

```
document.body.sayTagName = function() {  
    alert(this.tagName);  
};
```

```
document.body.sayTagName(); // BODY (la valeur de "this" dans la méthode
```

tous les attributs sont accessibles en utilisant les méthodes suivantes :

- **elem.getAttribute(name)** – vérifie l'existence.
- **elem.getAttribute(name)** – obtient la valeur.
- **elem.setAttribute(name, value)** – définit la valeur.
- **elem.removeAttribute(name)** – supprime l'attribut.

Attribus

208

On peut aussi lire tous les attributs en utilisant `elem.attributes` : une collection d'objets qui appartient à une classe intégrée avec `name` et la propriété `value`.

1. `getAttribute('About')` – la première lettre est en majuscules ici, et en HTML tout est en minuscules. Mais cela n'a pas d'importance: les noms d'attribut ne sont pas sensibles à la casse.
2. Nous pouvons assigner n'importe quoi à un attribut, mais il devient une chaîne. Nous avons donc ici "123" comme valeur.
3. Tous les attributs, y compris ceux que nous définissons, sont visibles dans `outerHTML`.
4. La collection `attributes` est itérable et possède tous les attributs de l'élément (standard et non standard) en tant qu'objets avec les propriétés `name` et `value`.

```
<body>
<div id="elem" about="Elephant"></div>

<script>
alert( elem.getAttribute('About') ); // (1) 'Elephant', lecture

elem.setAttribute('Test', 123); // (2), écriture

alert( elem.outerHTML ); // (3), voir si l'attribut est en HTML (oui)

for (let attr of elem.attributes) { // (4) lister tout
  alert(` ${attr.name} = ${attr.value}`);
}
</script>
</body>
```

Mais il y a des exclusions, par exemple `input.value` se synchronise uniquement de l'attribut → vers la propriété, mais pas dans l'autre sens :

```
<input>

<script>
let input = document.querySelector('input');

// attribute => property
input.setAttribute('id', 'id');
alert(input.id); // id (mis à jour)

// property => attribute
input.id = 'newId';
alert(input.getAttribute('id'));// newId (mis à jour)
</script>
```

```
<input>

<script>
let input = document.querySelector('input');

// attribute => property
input.setAttribute('value', 'text');
alert(input.value); // text

// NOT property => attribute
input.value = 'newValue';
alert(input.getAttribute('value'));// text (non mis à jour !)
</script>
```

Les propriétés DOM sont typées

209

Les propriétés DOM ne sont pas toujours des chaînes de caractères. Par exemple :

L'attribut **style** est une chaîne de caractères, mais la propriété **style** est un objet :

```
<div id="div" style="color:red;font-size:120%">Hello</div>

<script>
// string
alert(div.getAttribute('style')); // color:red;font-size:120%

// object
alert(div.style); // [object CSSStyleDeclaration]
alert(div.style.color); // red
</script>
```

la propriété **input.checked** (pour les cases à cocher) est un booléen :

```
<input id="input" type="checkbox" checked> checkbox

<script>
alert(input.getAttribute('checked')); // la valeur d'attribut
alert(input.checked); // la valeur de la propriété est : true
</script>
```

Modification du document

210

Pour créer des nœuds DOM:

`document.createElement(tag)`.

```
let div = document.createElement('div');
```

Crée un nouveau *nœud texte* avec le texte donné :

`document.createTextNode(text)`

```
let textNode = document.createTextNode('Here I am');
```

La plupart du temps, nous devons créer des nœuds d'élément, tels que le `div` pour le message

Pour faire apparaître la `div`, nous devons l'insérer quelque part dans `document`. Par exemple, dans l'élément `<body>`, référencé par `document.body`.

Il existe une méthode spéciale `append` pour cela : `document.body.append(div)`.

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  let div = document.createElement('div');
  div.className = "alert";
  div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";

  document.body.append(div);
</script>
```

Append,Prepend,After,Before

211

Voici plus de méthodes d'insertion, elles spécifient différents endroits où insérer :

- `node.append(...nodes or strings)` – ajouter des nœuds ou des chaînes de caractères à la fin de node,
- `node.prepend(...nodes or strings)` – insérer des nœuds ou des chaînes de caractères au début de node,
- `node.before(...nodes or strings)` -- insérer des nœuds ou des chaînes de caractères avant node,
- `node.after(...nodes or strings)` -- insérer des nœuds ou des chaînes de caractères après node,
- `node.replaceWith(...nodes or strings)` -- remplace node avec les nœuds ou chaînes de caractères donnés.

```

<ol id="ol">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  ol.before('before'); // insère la chaîne de caractères "before" avant <ol>
  ol.after('after'); // insère la chaîne de caractères "after" après <ol>

  let liFirst = document.createElement('li');
  liFirst.innerHTML = 'prepend';
  ol.prepend(liFirst); // insère liFirst au début de <ol>

  let liLast = document.createElement('li');
  liLast.innerHTML = 'append';
  ol.append(liLast); // insère liLast à la fin de <ol>
</script>

```

before
<ol id="ol">
 prepend
 0
 1
 2
 append

after

insertAdjacentHTML/Text/Element

212

Pour l'insertion nous pouvons utiliser une autre méthode assez polyvalente : `elem.insertAdjacentHTML(where, html)`. Le premier paramètre est un mot de code, spécifiant où insérer par rapport à `elem`. Doit être l'un des suivants :

- "**beforebegin**" – insère `html` immédiatement avant `elem`,
- "**afterbegin**" – insère `html` dans `elem`, au début,
- "**beforeend**" – insère `html` dans `elem`, à la fin,
- "**afterend**" – insère `html` immédiatement après `elem`.

Le second paramètre est une chaîne HTML insérée "au format HTML".

```
<p>Hello</p>
<div id="div"></div>
<p>Bye</p>
```

```
<div id="div"></div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p>Hello</p>');
  div.insertAdjacentHTML('afterend', '<p>Bye</p>');
</script>
```

La méthode a deux sœurs :

- **`elem.insertAdjacentText(where, text)`** : la même syntaxe, mais une chaîne de caractères `text` est insérée en tant que texte au lieu de HTML,
- **`elem.insertAdjacentElement(where, elem)`** – la même syntaxe, mais insère un élément.

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: 4px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
  document.body.insertAdjacentHTML("afterbegin", `<div class="alert">
    <strong>Hi there!</strong> You've read an important message.
  </div>`);
</script>
```

Suppression de nœuds

213

Pour supprimer un nœud, il existe une méthode `node.remove()`. Faisons disparaître notre message après une seconde :

: si nous voulons **déplacer** un élément vers un autre endroit – il n'est pas nécessaire de le supprimer de l'ancien.

```
<div id="first">First</div>
<div id="second">Second</div>
<script>
  // pas besoin d'appeler remove
  second.after(first); // prend #second et après insère #first
</script>
```

```
<style>
.alert {
  padding: 15px;
  border: 1px solid #d6e9c6;
  border-radius: px;
  color: #3c763d;
  background-color: #dff0d8;
}
</style>

<script>
let div = document.createElement('div');
div.className = "alert";
div.innerHTML = "<strong>Hi there!</strong> You've read an important message.";

document.body.append(div);
setTimeout(() => div.remove(), 1000);
</script>
```

L'appel `elem.cloneNode(true)` crée un clone “profond” de l'élément avec tous les attributs et sous-éléments.

- Si nous appelons `elem.cloneNode(false)`,
- alors le clone est fait sans éléments enfants.

```
let div2 = div.cloneNode(true); // clone the message
div2.querySelector('strong').innerHTML = 'Bye there!'; // change le clone

div.after(div2); // affiche le clone après le div existant
```

Document Fragment

214

DocumentFragment est un nœud DOM spécial qui sert de wrapper pour passer autour des listes de nœuds. Nous pouvons y ajouter d'autres nœuds, mais lorsque nous l'insérons quelque part, son contenu est inséré à la place. Par exemple, la fonction **getListContent** ci-dessous génère un fragment avec des éléments ****, qui sont ensuite insérés dans **** :

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let fragment = new DocumentFragment();

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    fragment.append(li);
  }

  return fragment;
}

ul.append(getListContent()); // (*)
</script>
```

```
<ul>
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
```

DocumentFragment est rarement utilisé explicitement

```
<ul id="ul"></ul>

<script>
function getListContent() {
  let result = [];

  for(let i=1; i<=3; i++) {
    let li = document.createElement('li');
    li.append(i);
    result.push(li);
  }

  return result;
}

ul.append(...getListContent()); // append + "..." operator = friends!
</script>
```

Méthodes à l'ancienne

215

Il existe également des méthodes de manipulation du DOM “à l’ancienne”, qui existent pour des raisons historiques. Ces méthodes viennent d’une époque très ancienne. De nos jours, il n’y a aucune raison de les utiliser, depuis qu’il existe des méthodes modernes, telles que append, prepend, before, after, remove, replaceWith, qui sont plus flexibles.

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = 'Hello, world!';

  list.appendChild(newLi);
</script>
```

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>

<script>
  let li = list.firstElementChild;
  list.removeChild(li);
```

```
<ol id="list">
  <li>0</li>
  <li>1</li>
  <li>2</li>
</ol>
<script>
  let newLi = document.createElement('li');
  newLi.innerHTML = 'Hello, world!';

  list.insertBefore(newLi, list.children[1]);
</script>
```

Pour insérer newLi comme premier élément, nous pouvons le faire comme ceci :

```
list.insertBefore(newLi,
list.firstChild);
```

Pour remplacer: .replaceChild(newElem, node)

L’appel à `document.write` ne fonctionne que pendant le chargement de la page.

Styles et classes

216

JavaScript peut modifier les classes et les propriétés de style.

Nous devons toujours favoriser l'utilisation des classes CSS plutôt que style.

Ce dernier devrait seulement être utilisé si les classes sont incapables d'effectuer la tâche requise.

Si nous attribuons quelque chose à elem.className, elle remplace la chaîne entière de classes. Parfois c'est ce que nous avons besoin, mais souvent, nous voulons seulement ajouter ou enlever une classe.

Il y a une autre propriété pour ce besoin: elem.classList.

elem.classList est un objet spécial avec des méthodes pour add/remove/toggle une seule classe.

Méthodes de classList:

- elem.classList.add/remove("class") – ajoute ou enlève la classe.
- elem.classList.toggle("class") – ajoute la classe si elle n'existe pas, sinon enlève-la.
- elem.classList.contains("class") – vérifie pour la classe donnée, renvoie true/false.

```
<body class="page d'accueil">
<script>
  alert(document.body.className); // page d'accueil
</script>
</body>
```

```
<body class="page d'accueil">
<script>
  // ajouter une classe
  document.body.classList.add('article');

  alert(document.body.className); // l'article de la page d'accueil
</script>
</body>
```

Style de l'élément

217

La propriété `elem.style` est un objet qui correspond à ce qui est écrit dans l'attribut "style". Attribuant `elem.style.width="100px"` fonctionne de la même façon qu'un attribut `style` ayant une chaîne `width:100px`.

```
document.body.style.backgroundColor = prompt('background color?', 'green');
```

Les propriétés prédéterminées par le navigateur comme `-moz-border-radius`, `-webkit-border-radius` suivent la même règle: un tiret signifie une majuscule.
Prenons, par exemple:

```
button.style.MozBorderRadius = '5px';
button.style.WebkitBorderRadius = '5px';
```

Parfois nous voulons attribuer une propriété de style, et ensuite la retirer.

Par exemple, pour cacher un élément, nous pouvons définir `elem.style.display = "none"`.

Plus tard, nous voulons peut-être enlever `style.display` comme si cette propriété n'était définie.

Au lieu de `delete elem.style.display`, nous devons attribuer une chaîne vide à la propriété de `style`: `elem.style.display = ""`.

```
// si nous exécutons cette code, <body> clignotera
document.body.style.display = "none"; // cacher
setTimeout(() => document.body.style.display = "", 1000); // retour à la normale
```

Pour définir un style complet comme une chaîne, il y a une propriété spéciale `style.cssText`

```
<div id="div">Bouton</div>

<script>
  // nous pouvons attribuer des drapeaux de style spéciaux comme "important" ici
  div.style.cssText=`color: red !important;
    background-color: yellow;
    width: 100px;
    text-align: center;
  `;

  alert(div.style.cssText);
</script>
```

Ajouter les valeurs des unités au style

ajouter des unités de CSS aux valeurs.

Par exemple, nous ne devrions pas attribuer elem.style.top à 10, mais plutôt à 10px. Sinon ça ne fonctionnera pas:

```
<body>
<script>
    // ne fonctionne pas!
    document.body.style.margin = 20;
    alert(document.body.style.margin); // '' (chaîne vide, l'affectation est ignorée)

    // maintenant ajoutez l'unité de CSS (px) - et ça fonctionne
    document.body.style.margin = '20px';
    alert(document.body.style.margin); // 20px

    alert(document.body.style.marginTop); // 20px
    alert(document.body.style.marginLeft); // 20px
</script>
</body>
```

Styles calculés

219

nous voulons savoir la taille, les marges et la couleur d'un élément. Comment faire?

Il y a une méthode pour cela: **getComputedStyle**.

element

Élément pour lire la valeur de.

pseudo

Un pseudo-élément si nécessaire, par exemple ::before. Une chaîne vide ou aucun argument signifie l'élément lui-même.

```
getComputedStyle(element, [pseudo])
```

```
1 <head>
2   <style> body { color: red; margin: 5px } </style>
3 </head>
4 <body>

5   <script>
6     let computedStyle = getComputedStyle(document.body);

7     // maintenant nous pouvons en lire la marge et la couleur
8
9       alert( computedStyle.marginTop ); // 5px
10      alert( computedStyle.color ); // rgb(255, 0, 0)
11
12    </script>
13
14 </body>
```

Introduction aux événements du navigateur

220

mohamed@goumih.com

- ▶ Un événement est un signal que quelque chose s'est produit. Tous les nœuds DOM génèrent de tels signaux (mais les événements ne sont pas limités au DOM). Voici une liste des événements DOM les plus utiles, juste pour jeter un coup d'œil :
- ▶ Pour réagir aux événements, nous pouvons attribuer un gestionnaire - une fonction qui s'exécute en cas d'événement. Les gestionnaires sont un moyen d'exécuter du code JavaScript en cas d'actions de l'utilisateur. Il existe plusieurs façons d'affecter un gestionnaire. Voyons-les, en commençant par la plus simple.

```
<input value="Click me" onclick="alert('Click!')" type="button">
```

```
<script>
function count() {
  for(let i=1; i<=3; i++) {
    alert(" number " + i);
  }
}
</script>
<input type="button" onclick="count()" value="Compter Nombre">
</body>
```

Evènements Dom

221

Nous pouvons affecter un gestionnaire à l'aide d'une propriété DOM sur les éléments en utilisant des fonctions.

```
<input id="elem" type="button" value="Click me">
<script>
  elem.onclick = function() {
    alert('Thank you');
  };
</scrint>
```

addEventListener :permet de ajouter un évènement

removeEventListener :permet de supprimer un évènement

```
element.addEventListener("click", clickFct); //Événement : clic de la souris
element.addEventListener("mouseover", mouseoverFxn); //Événement : passage de la souris sur un élément
element.addEventListener("mouseout", mouseoutFxn); //Événement : la souris quitte l'élément
```

```
function handler() {
  alert( 'Thanks!' );
}
```

```
input.addEventListener("click", handler);
// ....
input.removeEventListener("click", handler);
```

```
<input id="elem" type="button" value="Click me"/>

<script>
  function handler1() {
    alert('Thanks!');
  };

  function handler2() {
    alert('Thanks again!');
  }
```

```
elem.onclick = () => alert("Hello");
elem.addEventListener("click", handler1); // Thanks!
elem.addEventListener("click", handler2); // Thanks again!
```

Actions par défaut du navigateur

222

Si nous gérons un évènement avec Javascript, nous pouvons ne pas avoir envie de déclencher l'action de navigateur associée, et déclencher un autre comportement à la place.

L'évènement continue sa propagation habituelle à moins qu'un des gestionnaires d'évènement invoque [stopPropagation\(\)](#) ou [stopImmediatePropagation\(\)](#) pour interrompre la propagation.

Il y a deux manières de dire au navigateur que nous ne souhaitons pas qu'il agisse:

- La manière principale est d'utiliser l'objet **event**. Il y a une méthode **event.preventDefault()**.
- Si le gestionnaire d'évènement a été assigné en utilisant **on<event>** (pas par **addEventListener**), alors renvoyer **false** fonctionne de la même manière.

Dans cet HTML un clic sur un lien n'entraîne pas une navigation, le navigateur ne fait rien :

```
1 <a href="/" onclick="return false">Cliquez ici</a>
2 ou
3 <a href="/" onclick="event.preventDefault()">ici</a>
```

[Cliquez ici](#) ou [ici](#)

```
<ul id="menu" class="menu">
  <li><a href="/html">HTML</a></li>
  <li><a href="/javascript">JavaScript</a></li>
  <li><a href="/css">CSS</a></li>
</ul>
<script>
menu.onclick = function(event) {
  if (event.target.nodeName != 'A') return;

  let href = event.target.getAttribute('href');
  alert( href );

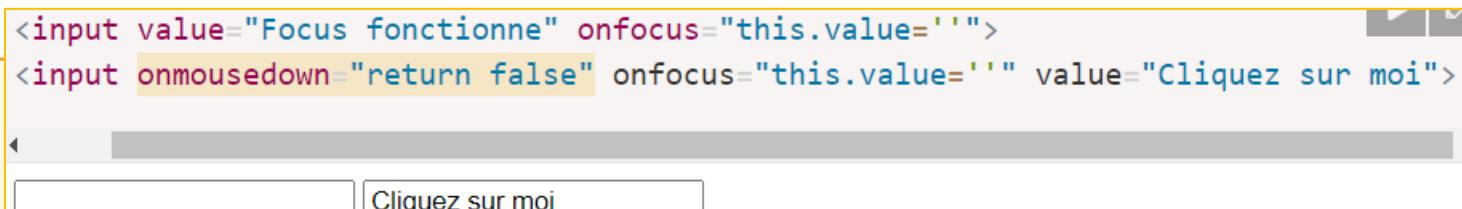
  return false;
};
```

Prevent Default

223

Certains évènements se suivent les uns après les autres. Si nous empêchons le premier évènement, il n'y aura pas de second. Par exemple, mousedown sur un champ <input> entraîne un focus dessus, et l'évènement focus. Si nous empêchons l'évènement mousedown, il n'y a pas de focus.

Essayez de cliquer sur le premier <input> ci-dessous – l'évènement focus se produit. Mais si vous cliquez sur le second, il n'y a pas de focus.



```
<input value="Focus fonctionne" onfocus="this.value=''">
<input onmousedown="return false" onfocus="this.value=''" value="Cliquez sur moi">
```

`preventDefault()` : signale au navigateur que ce gestionnaire n'appellera pas .
`event.stopPropagation()` arrête la propagation
La propriété `event.defaultPrevented` est true si l'action par défaut a été empêchée, et false dans les autres cas.

```
<p>Clic droit ici pour le menu contextuel du document (une vérification a été faite)</p>
<button id="elem">Clic droit ici pour le menu contextuel du bouton</button>

<script>
elem.oncontextmenu = function(event) {
  event.preventDefault();
  alert("Menu contextuel du bouton");
};

document.oncontextmenu = function(event) {
  if (event.defaultPrevented) return;

  event.preventDefault();
  alert("Menu contextuel du document");
};
</script>
```

PreventDefault :Exemple

224

Empêcher le clique sur un checkbox:

Essayez de cliquer sur la case à cocher.

Checkbox:

Désolé ! preventDefault() ne vous laissera pas cocher ceci.

OK

<p>Essayez de cliquer sur la case à cocher.</p>

```
<form>
  <label for="id-checkbox">Checkbox:</label>
  <input type="checkbox" id="id-checkbox"/>
</form>
```

```
<script>
document.querySelector("#id-checkbox").
addEventListener("click", function(event) {
  alert("Désolé ! preventDefault() ne vous laissera pas cocher ceci.");
  event.preventDefault();
}, false);
```

Empêcher les pressions du clavier sur un champ texte

```
<form>
  <label>Un champ texte
  <input type="text" id="mon-champ-texte">
</label>
</form>

<script>
let monChampTexte = document.getElementById('mon-champ-texte');
monChampTexte.addEventListener('keypress', bloqueSaisie, false);

function bloqueSaisie(evt) {
  evt.preventDefault();
  console.log("Une saisie a été empêchée.");
};
```

Evènements de la souris

mohamed@goumih.com

225

Les types d'évènements de Souris

Nous avons déjà vu certains de ces événements :

mousedown/mouseup:Le bouton de la souris est appuyé puis relâché sur un élément.

mouseover/mouseout:Le pointeur de la souris entre ou sort d'un élément.

Mousemove :Chaque déplacement de la souris sur un élément déclenche cet évènement.

Click:est déclenché après un évènement mousedown et suite à un mouseup sur le même élément, si le bouton gauche de la souris a été utilisé

Dblclick:Se déclenche après deux clics sur le même élément dans un court laps de temps. Rarement utilisé de nos jours.

Contextmenu:Se déclenche lorsque le bouton droit de la souris est enfoncé. Il existe d'autres façons d'ouvrir un menu contextuel,

par ex. en utilisant une touche spéciale du clavier, il se déclenche dans ce cas également, donc ce n'est pas exactement l'événement de la souris.

... Il y a aussi plusieurs autres événements,

Ordre des événements

Comme vous pouvez le voir dans la liste ci-dessus, une action utilisateur peut déclencher plusieurs événements.

Par exemple, un clic gauche déclenche d'abord **mousedown**, lorsque le bouton est enfoncé, puis **mouseup** et **click** lorsqu'il est relâché.

Au cas où une action unique initialise plusieurs événements, leur ordre est fixé. C'est-à-dire que les gestionnaires sont appelés dans l'ordre **mousedown → mouseup → click**.

Evènements souris

226

Empêcher la sélection sur le mousedown

Le double clic de souris a un effet secondaire qui peut être dérangeant dans certaines interfaces: il sélectionne du texte.

par exemple, double-cliquer sur le texte ci-dessous le sélectionne en plus de notre gestionnaire :

```
<span ondblclick="alert('dblclick')">Double-click me</span>
```

Si on appuie sur le bouton gauche de la souris et, sans le relâcher, on déplace la souris, la sélection devient alors souvent indésirable.

Il existe plusieurs façons d'empêcher la sélection, que vous pouvez lire dans le chapitre [Selection et Range](#).

Dans ce cas particulier, le moyen le plus raisonnable consiste à empêcher l'action du navigateur lors du mousedown. Il empêche ces deux sélections :

Avant...

```
<b ondblclick="alert('Click!')" onmousedown="return false">
```

Double-click sur moi

```
</b>
```

...Apres

Forms:

227

Les formulaires de document sont membres de la collection spéciale **document.forms**. C'est ce qu'on appelle une « collection nommée » : elle est à la fois nommée et ordonnée. Nous pouvons utiliser à la fois le nom ou le numéro dans le document pour obtenir le formulaire.

```
<form name="my">
  <input name="one" value="1">
  <input name="two" value="2">
  <input type="radio" name="age" value="10">
  <input type="radio" name="age" value="20">
</form>
<script>
  // obtenir le formulaire
  let form = document.forms.my; // <form name="my"> element
  let form1 = document.forms[0]; // le premier form dans la page
  // Obtenez l'élément
  let elem = form.elements.one; // <input name="one"> element
  alert(elem.value); // 1
  // Il peut y avoir plusieurs éléments portant le même nom.
  // Ceci est typique des boutons radio et des cases à cocher.
  // Dans ce cas, form.elements[nom] est une collection.
  let ageElems = form.elements.age; // la déclaration résumé form.age sans elements
  alert(ageElems[0].value); //10
```

Récupérer les informations d'un formulaire

228

Ici on a beaucoup de formes on utilise pour chaque form un **id**

```
<form id="f1">
  <input type="button" id="i1" onclick="alert( 'je suis '+
    document.getElementById('i1').value+' mon id est :'+document.forms[0].id);"
    value="Form1" />
</form>

<form id="f2">
  <input type="button" id="i2" onclick="alert( 'je suis '+
    document.getElementById('i2').value+' mon id est :'+document.forms[1].id);"
    value="Form2" />
</form>

<form id="f3">
  <input type="button" id="i3" onclick="alert( 'je suis '+
    document.getElementById('i3').value+' mon id est :'+document.forms[2].id);"
    value="Form3" />
</form>
```

Accéder aux formulaires nommés

229

```
<form name="login">
  <input name="email" type="email">
  <input name="password" type="password">
  <button type="submit">Login</button>
</form>

<script>
  var loginForm = document.forms.login; // Ou document.forms['login']
  loginForm.elements.email.placeholder = 'test@example.com';
  loginForm.elements.password.placeholder = 'password';
</script>
```

Forms :select ,options

230

exemples de la façon d'obtenir des valeurs sélectionnées à partir d'une seule et multi-sélection :

```
<select id="select">
  <option value="apple">Apple</option>
  <option value="pear">Pear</option>
  <option value="banana">Banana</option>
</select>

<script>
  select.options[2].selected = true;
  select.selectedIndex = 2;

  alert(select.value );
</script>
```

```
<select id="select2" multiple>
  <option value="blues" selected>Blues</option>
  <option value="rock" selected>Rock</option>
  <option value="classic">Classic</option>
</select>

<script>
  let selected = Array.from(select2.options)
    .filter(option => option.selected)
    .map(option => option.value);

  alert(selected); // blues,rock
</script>
```

, il y a une belle syntaxe courte pour créer un élément option :

```
<select id="genres">
  <option value="rock">Rock</option>
  <option value="blues" selected>Blues</option>
</select>
```

```
let option = new Option("Text", "value");
// creates <option value="value">Text</option>

<script>
  // 1)
  let selectedOption = genres.options[genres.selectedIndex];
  alert( selectedOption.value );

  // 2)
  let newOption = new Option("Classic", "classic");
  genres.append(newOption);

  // 3)
  newOption.selected = true;
</script>
```

onFocus/onblur

231

L'évènement **focus** est appelé lors du focus et **blur** lorsque l'élément perds le focus.

Utilisons les pour la validation d'un champ de saisie.

Dans l'exemple ci-dessous: Le gestionnaire blur vérifie si l'adresse mail est entrée, et sinon – affiche une erreur.

- Le gestionnaire focus masque le message d'erreur (au moment de blur le champ sera vérifié à nouveau):

```
<style>
  .error {
    background: red;
  }
</style>

Entrez votre email: <input type="email" id="input">
<input type="text" style="width:220px"
placeholder="entrez une adresse email invalide
et essayez de mettre le focus sur ce champ">

<script>
  input.onblur = function() {
    if (!this.value.includes('@')) { // pas une adresse email
      // affiche l'erreur
      this.classList.add("error");
      // ...et remet le focus
      input.focus();
    } else {
      this.classList.remove("error");
    }
  };
</script>
```

```
<style>
  .invalid { border-color: red; }
  #error { color: red }
</style>
Entrez votre email: <input type="email" id="input">
<div id="error"></div>

<script>
input.onblur = function() {
  if (!input.value.includes('@')) { // pas une adresse email
    input.classList.add('invalid');
    error.innerHTML = 'Veuillez entrer une adresse email valide.';
  }
};

input.onfocus = function() {
  if (this.classList.contains('invalid')) {
    // retire l'erreur
    this.classList.remove('invalid');
    error.innerHTML = "";
  }
};
</script>
```

Onfocus: focusin/focusout

232

```
<!-- lors du focus sur le formulaire -- ajouter la classe -->
<form
  onfocus="this.className='focused'"
    <input onfocus="this.value=''" type="text" id="n" name="name" value="Name">
    <input type="text" onfocus="this.value=''" name="surname" value="Surname">
</form>

<style> .focused { outline: 1px solid red; } </style>
```

```
<form id="form">
  <input type="text" name="name" value="Name">
  <input type="text" name="surname" value="Surname">
</form>

<style> .focused { outline: 1px solid red; } </style>

<script>
  form.addEventListener("focusin", () => form.classList.add('focused'));
  form.addEventListener("focusout", () => form.classList.remove('focused'));
</script>
```

onchange ,oninput, onpaste, oncopy

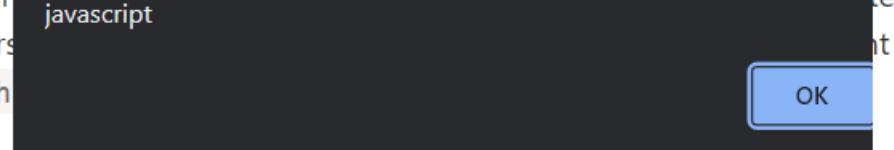
233

L'événement **onchange** se déclenche lorsque le changement de la valeur de l'élément a fini de se réaliser.

```

1 <select onchange="alert(this.value)">
2   <option value="">Select something</option>
3   <option value="1">Option 1</option>
4   <option value="2">Option 2</option>
5   <option value="3">Option 3</option>
6 </select>
```

Option 2 ▾



```

1 <input type="text" onchange="alert(this.value)">
2 <input type="button" value="Button">
```

javascript

OK

javascrip| Button

Événements: cut, copy, paste

Ces événements se produisent lors de la coupe/copie/collage d'une valeur.

```

<input type="text" id="input">
<script>
  input.onpaste = function(event) {
    alert("paste: " + event.clipboardData.getData('text/plain'));
    event.preventDefault();
  };

  input.oncut = input.oncopy = function(event) {
    alert(event.type + '-' + document.getSelection());
    event.preventDefault();
  };
</script>
```

L'événement **input** se déclenche à chaque fois qu'une valeur est modifiée par l'utilisateur.

```

1 <input type="text" id="input" oninput: <span id="result"></span>
2 <script>
3   input.oninput = function() {
4     result.innerHTML = input.value;
5   };
6 </script>
```

d oninput: d

les évènements keydown et keyup

234

Les évènements keydown surviennent lorsqu'une touche est appuyée, et ensuite intervient keyup – lorsqu'elle est relâchée.

```
document.addEventListener('keydown', function(event) {  
    if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {  
        alert('Undo!')  
    }  
});
```

Par exemple, la balise <input> en bas s'attend à recevoir un numéro de téléphone, alors elle n'accepte que les touches numériques, +, () ou bien -:

```
<input onkeydown="checkPhoneKey(event.key)" placeholder="Phone, please" type="tel">  
<script>  
    function checkPhoneKey(key) {  
        return (key >= '0' && key <= '9') || [ '+', '(', ')', '-' ].includes(key);  
    }  
</script>
```

onSubmit

235

L'événement `submit` se déclenche lorsque le formulaire est soumis, il est généralement utilisé pour valider le formulaire avant de l'envoyer au serveur ou pour abandonner la soumission et la traiter en JavaScript. La méthode `form.submit()` permet de lancer l'envoi de formulaire depuis JavaScript. Nous pouvons l'utiliser pour créer et envoyer dynamiquement nos propres formulaires au serveur.

```
<form id="FORM1" method="post" action="/code.php">
<label>Nom <input type="text" name="nom"></label><br>
<label>Age <input type="text" name="Age"><label> <br>
<input type="submit" onclick="SoumettreForm()" value="SUBMIT">
<input type="button" onclick="ResetForm()" value="RESET">
</form>
<p id="message"></p>
<script>
    function SoumettreForm() {
        document.getElementById("FORM1").submit();
    }
    function ResetForm() {
        document.getElementById("FORM1").reset();
        document.getElementById("message").innerHTML="Formulaire réinitialisé";
    }
</script>
```

Pour soumettre manuellement un formulaire au serveur, nous pouvons appeler `form.submit()`. Parfois, cela est utilisé pour créer et envoyer manuellement un formulaire, comme ceci:

```
<form onsubmit="alert('submit!');return false">
    First: Enter in the input field <input type="text" value="text"><br>
    Second: Click "submit": <input type="submit" value="Submit">
</form>
```

Enter in the input field text
nd: Click "submit": Submit

```
let form = document.createElement('form');
form.action = 'https://google.com/search';
form.method = 'GET';

form.innerHTML = '<input name="q" value="onsubmit javascript">';

// le formulaire doit être dans le document pour le soumettre
document.body.append(form);

form.submit();
```

Validation du formulaire

236

JavaScript est souvent utilisé pour valider les forms:

- Si un champ de formulaire (fname) est vide, cette fonction alerte un message et renvoie false pour empêcher la soumission du formulaire :

```
<!DOCTYPE html>
<html>
<head>
<script>
function validateForm() {
    let x = document.forms["myForm"]["fname"].value;
    if (x == "") {
        alert("Name must be filled out");
        return false;
    }
}
</script>
</head>
<body>

<h2>JavaScript Validation</h2>
|
<form name="myForm" action="/action_page.php" onsubmit="return validateForm()"
method="post">
    Name: <input type="text" name="fname">
    <input type="submit" value="Submit">
</form>

</body>
</html>
```

Validation des données

237

Exemple : Si un champ de formulaire (nom) est vide, la fonction affiche un message et renvoie false pour empêcher la soumission du formulaire

```
function validerForm() {  
    let x = document.forms["myForm"]["nom"].value;  
    if (x == "") {  
        alert("Le champ \"nom\" doit être saisi");  
        return false;  
    }  
}
```

```
<form name="myForm" action="/code.php" onsubmit="return validerForm()" method="post">  
    Nom: <input type="text" name="nom">  
    <input type="submit" value="Submit">  
</form>
```

TP7:

238

- ▶ Créer l'application suivante qui permet d'enregistrer des éléments dans **localStorage** et l'affiche dans une liste.

stagiaire

Ajouter un stagiaire

- ▶ Créer l'application suivante qui permet d'enregistrer des éléments dans **localStorage** et l'affiche dans une liste.

Ajouter un élément :

3

Ajouter

La Liste :

- 1-elment 1
- 2-java
- 3-php

EP33:

239

Mettez tous les liens externes en orange en modifiant leur propriété style.

Un lien est externe si :

- Son href contient ://
- Mais ne commence pas par http://google.com.

- <http://google.com>
- <tutorial.html>
- <local/path>
- <ftp://ftp.com/my.zip>
- <http://nodejs.org>
- <http://internal.com/test>

Créer une fonction **randomColor** qui permet de générer un **backgroundcolor** aléatoire lorsque on clique sur le button changer couleur.

Par la suite ajouter un autre **button** et appliquer la fonction à tous les buttons

Changer Couleur

Écrivez une fonction **Notification({top = 0, right = 0, className, html})** qui crée une notification : avec le contenu donné. La notification devrait disparaître automatiquement après 1,5 seconde.

```
Notification({ top: 10, right: 10, html: 'Hello ' + i++, className: "welcome" });
```

Notification is on the right

Hello 12

Lore ipsum dolor sit amet, consectetur adipisicing elit. Dolorum aspernatur quam ex eaque inventore quod voluptatem adipisci omnis nemo nulla fugit iste numquam ducimus cumque

EP34:

240

Créez une fonction **Clean(element)** qui supprime tout de l'élément.

```
<ol id="element">
<li>Hello</li>
<li>World</li>
</ol>
```

Écrivez une programme qui demande de créer une liste à partir des entrées utilisateur.

Pour chaque élément de la liste :

1. Interrogez un utilisateur sur son contenu en utilisant `prompt`.
2. Créez le `` avec et ajoutez-le à ``.
3. Continuez jusqu'à ce que l'utilisateur annule l'entrée (en appuyant sur la touche `Esc` ou une entrée vide).

Tous les éléments doivent être créés dynamiquement.

Si un utilisateur tape des balises HTML, elles doivent être traitées comme un texte.

Enter le texte à écrire dans la liste

Mohamed

OK

Annuler

Enter le texte à écrire dans la liste

Ahmed

OK

Annuler

- Mohamed
- Ahmed

Écrivez une fonction **createArbre** qui crée une liste imbriquée `ul/li` à partir de l'objet imbriqué.

```
let personnes =
{
  "Filles": {
    "Fatima": {},
    "fadma": {}
  },
  "Garcons": {
    "mohamed": {
      "Hicham": {},
      "Rachid": {}
    },
    "Prof": {
      "Mohamed": {},
      "Jawad": {}
    }
}
```

- Filles
 - Fatima
 - fadma
- Garcons
 - mohamed
 - Hicham
 - Rachid
 - Prof
 - Mohamed
 - Jawad

EP35:

241

1. Récupérer les titres de niveau h2 dans une page html et les faire apparaître comme une liste à puces. 1. Faire apparaître une image quelconque dans la page.
2. Ajouter un bouton qui permet de disparaître l'image et un bouton qui fait revenir l'image.
3. Programmer un effet de *fondu* pour que l'image apparaisse et disparaîsse progressivement(utiliser opacity setTimeout ou setInterval).
4. Provoquer cette fois un *flash* lorsque la souris passe sur une image : fondu au blanc rapide puis réapparition de l'image.
5. Placer des images dans un tableau JavaScript.
6. Créer une fonction de l'affichage aléatoire de l'une de ces images, toutes les deux secondes.et lorsque q'on clique sur deux buttons suivant précédent
7. Définir un tableau JS qui contienne un texte descriptif pour chacune des photos affichées.
8. quand la souris passe sur une image, fait apparaître le texte associé.
9. disparaître le texte lorsque la souris quitte l'image.

EP36:

242

```
<select id="genres">
  <option value="rock">Rock</option>
  <option value="blues" selected>Blues</option>
</select>
```

1. Affiche la valeur et le texte de l'option sélectionnée.
2. Ajoutez une option : Classique. Faites-le sélectionné.

Créer un script qui vérifier l'envoi d'un checkbox

La question est obligatoire : il faut au moins une réponse mais pas plus de 4.

Quelles sont vos glaces préférées ?
 Chocolat
 Fraise
 Pistache
 Vanille

Envoyer

Créer un jeu qui permet de créer des buttons avec des numéros et une case vide lorsque l'on clique sur un bouton, l'échange se fait avec la case vide.

1	11	8	5
7	10	4	6
13	9	2	14
15	12	3	

Créez un bouton qui se cache lui même au clic et un autre button pour disparaître le texte

Texte

Vous avez le html suivant créer un button pour supprimer Un div lorsqu'on clique sur le button étoile

```
<div>
  <div class="pane">
    <h3>Classe1</h3>
  </div>
  <div class="pane">
    <h3>Classe2</h3>
  </div>
  <div class="pane">
    <h3>Classe3</h3>
    <p>
    </p>
  </div>
</div>
```

[x]

Classe1

[x]

Classe2

[x]

Classe3

Réaliser l'application suivante :

- Le conteneur est défini avec une hauteur et une largeur fixes, et une propriété **overflow** de **auto** pour activer la barre de défilement.
- Les éléments de la liste sont stylisés avec une marge, un padding et une couleur de fond.
- L'élément actif est stylisé avec une couleur de fond différente.
- Le script récupère le conteneur et les éléments de la liste à partir du DOM.
- La fonction **setActiveItem** ajoute la classe "active" à l'élément actif et retire la classe "active" de l'élément précédent.
- L'écouteur d'événements pour le scroll calcule la position de scroll actuelle et met à jour l'élément actif en fonction de l'indice corresponds

```
<div id="conteneur">
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
    <li>Item 4</li>
    <li>Item 5</li>
    <li>Item 6</li>
  </ul>
</div>
```

```
#conteneur {
  width: 300px;
  height: 200px;
  overflow: auto;
}
li {
  list-style: none;
  padding: 10px;
  margin: 5px;
  background-color: #eee;
}
.active {
  background-color: #ccc;
```

Item 1

Item 2

Item 3

Item 4

EP38:

244

Créez une `<div>` qui se transforme en `<textarea>` lorsque l'on clique dessus.

La zone de texte permet de modifier le HTML présent dans la `<div>`.

Lorsque utilisateur clique sur Entrer .

ou perds le focus, la `<textarea>` se transforme à nouveau en `<div>`, et son contenu devient le HTML dans `<div>`.

je suis un div ,clic moi de me transformer en textarea

- Créer un formulaire html puis vérifier si les champs sont remplis avant d'envoyer les données si le cas afficher un message qui demande au utilisateur de les remplir

- Le formulaire doit être au centre de la fenêtre..
- Lorsque le formulaire est affiché, le focus doit être à l'intérieur de `<input>` pour l'utilisateur.

:Nom ff

Prenom:

Envoyer

vous devez remplir les 2 champs :!

Télécharger TP8
et
Réaliser les applications
Des 3 niveaux

Partie 9:

Les expressions régulières

Les expressions régulières

247

Les expressions régulières sont un moyen puissant de rechercher et de remplacer du texte.

En JavaScript, ils sont disponibles en tant que object [RegExp](#) et également intégrés dans les méthodes de chaînes de caractères.

Une expression régulière (également “**regexp**” ou simplement “**reg**”) est constituée d’un **pattern** et de **flags** optionnels.
Il existe deux syntaxes pour créer un objet expression régulière :

Les **slashes** sont utilisés lorsque nous connaissons l’expression régulière au moment de l’écriture du code .

Alors que **new RegExp** est plus utilisé lorsque nous devons créer une expression régulière “à la volée”

```
regexp = new RegExp("pattern", "flags");
```

Et la syntaxe courte, en utilisant des slash “/” :

```
regexp = /pattern/; // aucun marqueur
```

```
regexp = /pattern/gmi; // avec marqueurs g, m, et i
```

Flags : (Drapeaux) :

I : la recherche est insensible à la casse: pas de différence entre A et a

G:la recherche liste toutes les correspondances, sans lui ,seulement la première.

M:Mode multiligne.

S:correspondre au caractère de nouvelle ligne \n

U:Active le support complet Unicode. Le flag permet le traitement correct des paires de substitution.

Y:mode “Sticky” : chercher à la position exacte dans le texte

Les expressions régulières: méthode match

248

La méthode `str.match(regexp)` trouve tous les résultats de `regexp` dans la chaîne de caractères `str`. Il dispose de 3 modes de travail :

Si l'expression régulière a l'indicateur `g`, elle renvoie un tableau de toutes les correspondances :

Si `i` il renvoie le premier ,

Si aucun il renvoie **`undefined`** ou **`null`**

Si nous souhaitons que le résultat soit toujours un tableau, nous pouvons l'écrire comme ceci :

```
let matches = "JavaScript".match(/HTML/) || [];
console.log(matches); []
if (!matches.length) {
  console.log("pas de résultat");}  pas de résultat
```

```
let str = "dev digital ,dev digital";
console.log( str.match(/dev/gi) ); | [ 'dev', 'dev' ]
```

```
let result = str.match(/dev/i); // sans flag: g
console.log( result[0] ); // (premier occurence) dev
console.log( result[1] ); // (premier occurence) undefined
```

```
console.log( result.length ); 1
```

```
// Details:
console.log( result.index ); 0
console.log( result.input ); dev digital ,dev digital
```

```
let matches = "JavaScript".match(/HTML/);
console.log(matches); null
```

Les expressions régulières: méthode replace et test

La méthode **str.replace(regexp, replacement)** remplace les correspondances en utilisant regexp dans la chaîne de caractères str avec replacement (tous les résultats s'il y a un flag g, sinon seulement le premier).

Par exemple :

```
// sans g
console.log( "dev digital, dev digital".replace(/dev/i, "I study") ); | I study digital, dev digital
// avec g
console.log( "dev digital, dev digital".replace(/dev/ig, "I study") ); | I study digital, I study digital
```

Le deuxième argument est la chaîne de caractères replacement.

Nous pouvons utiliser des combinaisons de caractères spéciaux pour insérer des fragments de la correspondance :

Symboles Action dans la chaîne de caractères de remplacement string

\$& insère toute la correspondance match

\$` insère une partie de la chaîne de caractères avant la correspondance

\$' insère une partie de la chaîne de caractères après la correspondance

\$n si n est un nombre à un ou deux chiffres, alors il insère le contenu des nièmes parenthèses, plus à ce sujet dans le chapitre

\$<name> insère le contenu des parenthèses avec le `name` donné,

\$\$ insère le caractère \$

```
console.log( "J'aime HTML".replace(/HTML/, "$& et JavaScript") ); | J'aime HTML et JavaScript
```

La méthode **regexp.test(str)** recherche au moins une correspondance ; si elle est trouvée, retourne true, sinon false.

```
let l = "J'aime JavaScript";
let regexp = /j'aime/i;
console.log( regexp.test(l) ); | true
```

Les expressions régulières:

Classes de caractères

nous avons un numéro de téléphone tel que "+7(903)-123-45-67", et nous souhaitons le convertir en nombres purs : **79031234567**.

Pour ce faire, nous pouvons rechercher et supprimer tout ce qui n'est pas un nombre. **Les classes de caractères** peuvent nous aider.

Une classe de caractères est une notation spéciale qui correspond à n'importe quel symbole d'un certain ensemble.

Pour commencer, **explorons la classe “digit”**. Elle s'écrit comme `\d` et correspond à “n'importe quel chiffre”.

Par exemple, recherchons le premier chiffre dans le numéro de téléphone :

```
let str2 = "+7(903)-123-45-67";
let regexp2 = /\d/;

console.log( str2.match(regexp2) );  [ '7', index: 1,
```

1

```
let nT = "+7(903)-123-45-67";
let cc = /\d/g;
console.log( nT.match(cc) );  [ '7', '9', '0', '3', '1', '2', '3', '4', '5', '6', '7' ]
// Obtenons un numéro de téléphone composé uniquement de ces chiffres:
console.log( nT.match(cc).join('') );  79031234567
```

2

Les classes les plus utilisés sont :

\d (“d” vient de “digit” (“chiffre”)): un chiffre: un caractère de 0 à 9.

\s (“s” vient de “space” (“espace”)): Un symbole d'espace: inclut les espaces, les tabulations `\t`, les sauts de ligne `\n` et quelques autres caractères rares, tels que `\v`, `\f` et `\r`.

\w (“w” vient de “word” (“mot”)): Un caractère “verbeux”: soit une lettre de l'alphabet latin, soit un chiffre ou un trait de soulignement `_`.

Les lettres non latines (comme le cyrillique ou l'hindi) n'appartiennent pas au `\w`.

Par exemple, `\d\s\w` signifie un “chiffre” suivi d'un “caractère espace” suivi d'un “caractère verbeux”, tel que 1 a.

Les expressions régulières:

Classes de caractères

251

Une expression régulière peut contenir à la fois des symboles normaux et des classes de caractères.
Par exemple, `CSS\d` correspond à une chaîne `CSS` suivi d'un chiffre :

On peut également utiliser les classes de caractères :

```
console.log( " JS CSS HTML5!".match(/\s\w\w\w\w\d/) ); [ ' HTML5'
```

```
let string = "HTML CSS?";  
let reg = /CSS\d/  
console.log( string.match(reg) ); [ 'CSS3', index: 5,
```

Classes inverses:

Pour chaque classe de caractères, il existe une “classe inverse”, notée avec la même lettre, mais en majuscule.

L’“inverse” signifie qu’il correspond à tous les autres caractères, par exemple :

\D

Non-chiffre: tout caractère sauf \d, par exemple une lettre.

\S

Non-espace: tout caractère sauf \s, par exemple une lettre.

\W

Caractère non verbal : tout sauf \w, par exemple une lettre non latine ou un espace.

Au début du chapitre, nous avons vu comment créer un numéro de téléphone uniquement à partir d'une chaîne telle que +7(903)-123-45-67: trouver tous les chiffres et les concaténer.

Rechercher un motif non numérique \D et à le supprimer de la chaîne:

```
let strr = "+7(903)-123-45-67";  
console.log( strr.replace(/\D/g, "") ); 79031234567
```

Les expressions régulières: point

252

Un point . est une classe de caractères spéciale qui correspond à n'importe quel caractère sauf une nouvelle ligne.

Ou au milieu d'une expression régulière:

```
console.log( "DEV".match(/./) ); [ 'D', index: 0,
```

```
let css= /CS.4/;
console.log( "CSS4".match(css) ); [ 'CSS4', index: 0, input: 'CSS4',
console.log( "CS-4".match(css) ); [ 'CS-4', index: 0, input: 'CS-4',
console.log( "CS 4".match(css) ); [ 'CS 4', index: 0, input: 'CS 4'
```

A noter qu'un **point** signifie n'importe quel caractère, mais pas l'absence de caractère. Il doit y avoir un caractère avec lequel le faire correspondre :

```
console.log( "css".match(/CS.4/) ); null
// null, pas de correspondance
// car il n'y a pas de caractère pour le point
```

Le motif **[\s\S]** dit littéralement: "n'importe quoi".

Nous pourrions utiliser une autre paire de classes complémentaires, telles que **[\d\D]**
Ou même le **[^]** : car cela signifie correspondre à n'importe quel caractère sauf rien.

Par défaut, un **point** ne correspond pas au caractère de saut de ligne **\n**.

Par exemple, l'expression rationnelle A.B correspond à A, puis B avec n'importe quel caractère entre eux, sauf un saut de ligne **\n**, sinon on utilise **indicateur s**:

```
console.log( "A\nB".match(/A.B/) ); null
console.log( "A\nB".match(/A.B/s) ); //!avec s [ 'A\nB', index: 0, input: 'A\nB'
```

Pour nous, les chaînes **1-5** et **1 - 5** sont presque identiques. Mais si une expression régulière ne prend pas en compte les espaces, elle peut ne pas fonctionner. tous les caractères comptent, les espaces aussi.

```
console.log( "1 - 5".match(/\d-\d/) ); null
console.log( "1 - 5".match(/\d - \d/) ); [ '1 - 5', index: 0, input: '1 - 5',
// ou on peut utiliser la classe \s:
console.log( "1 - 5".match(/\d\s-\s\d/) ); [ '1 - 5', index: 0, input: '1 - 5'
```

Les expressions régulières:

Ancres :début ^ et fin \$

```
//!Par exemple, testons si le texte commence par dev
let str1 = "dev digital js";
console.log( /^dev/.test(str1) );  true
```

```
//!Similairement, nous pouvons vérifier si le texte
//!termine par js
console.log( /js$/.test(str1) );  true
```

```
//!Les deux ancre ensemble ^...$ 
/* Par exemple, pour vérifier si l'entrée de l'utilisateur est dans le bon format.
//?:Vérifions si une chaîne de caractères est une heure au format 12:34.
//todo: deux nombres, puis deux points, et enfin deux autres nombres :\d\d:\d\d:

let goodInput = "12:34";
let badInput = "12:345";

let ddd = /^\\d\\d:\\d\\d$/;
console.log( ddd.test(goodInput) );  true
console.log( ddd.test(badInput) );  false
//Ici, la correspondance pour \d\d:\d\d doit commencer juste après le début du texte ^,
// et la fin $ doit immédiatement suivre.
```

les ancre ne sont pas des caractères, mais des tests.

Les expressions régulières:

Mode multiligne des ancrés ^ \$, avec drapeau m

```
//!Dans l'exemple ci-dessous, le texte a plusieurs lignes.
//!Le motif / ^ \d / gm prend un chiffre du début de chaque ligne
```

```
let st = `

1st place: Ahmed
2eme place: Mohamed
3eme place: Bachir`;
```

```
console.log( st.match(/^\d/gm) );    [ '1', '2', '3' ]
/*Sans le drapeau M, seul le premier chiffre est apparié:
```

```
st = `1st place: Ahmed
2nd place: Mohamed
3rd place: Bachir`;
```

```
console.log( st.match(/^\d/g) );    [ '1' ]
//C'est parce que par défaut, un garet ne correspond qu'au début du texte,
//et en mode multiline - au début de n'importe quelle ligne.
```

```
//!«Début d'une ligne» immédiatement après une pause de ligne
//!Le test ^ en mode multiligne correspond à toutes les positions
//!précédés d'un caractère \n.
//!Le signe du dollar $ se comporte de la même manière.
//!L'expression régulière \d$ trouve le dernier chiffre dans chaque ligne
```

```
st= `ahmed: 1
Mohamed: 2
Bachir: 3`;
console.log( st.match(/^\d/g) );    null
console.log( st.match(/(\d$/gm) );   [ '1', '2', '3' ]
console.log( st.match(/(\d\n/g) );   [ '1\n', '2\n' ]
```

Les expressions régulières:

Limite de mot : \b

mohamed@goumih.com

255

Une limite de mot **\b** teste une position, de la même manière que les ancreς ^ et \$.
Le motif **\b**, il vérifie si la position dans la chaîne de caractères est une limite de mot.
Il y a trois positions possibles pour une limite de mot :

- **Au début de la chaîne de caractères**: si le premier caractère est alphanumérique **\w**.
- **Entre deux caractères d'une chaîne** : si seulement l'un des caractères correspond au motif **\w**.
- **À la fin de la chaîne de caractères**: si le dernier caractère correspond au motif **\w**.

Par exemple l'expression régulière **\bJava\b** sera trouvé dans **Hello, Java!**, où Java est un mot isolé, mais pas dans **Hello, JavaScript!**.

```
console.log( "Hello, Java!".match(/\bJava\b/) ); [ 'Java', index: 7,
console.log( "Hello, JavaScript!".match(/\bJava\b/) ); null
```

Dans la chaîne **Hello, Java!** les positions suivantes correspondent au motif **\b**:

Cette chaîne passe le test du motif **\bHello\b**, car :

1. Le début de la chaîne passe le premier test **\b**.

2. Puis trouve le mot **Hello**.

3. Enfin le test **\b** est encore valide, comme nous sommes entre 0 et une virgule.

Donc le motif **\bHello\b** sera trouvé, mais pas **\bHell\b** (car il n'y a pas de limite de mot après 1) ni **Java!\b** (car le point d'exclamation ne correspond pas au motif **\w**, il n'est donc pas suivi par une limite de mot).

```
console.log( "Hello, Java!".match(/\bHello\b/) ); [ 'Hello', index: 0,
console.log( "Hello, Java!".match(/\bJava\b/) ); [ 'Java', index: 7,
console.log( "Hello, Java!".match(/\bHell\b/) ); null
console.log( "Hello, Java!".match(/\bJava!\b/) ); null
```

```
console.log( "1 23 456 78".match(/\b\d\d\b/g) ); [ '23', '78' ]
console.log( "12,34,56".match(/\b\d\d\b/g) ); [ '12', '34', '56' ]
```

Le test de limite de mot **\b** vérifie qu'il doit y avoir **\w** d'un côté de la position et "not **\w**" – de l'autre côté.
Comme **\w** signifie a-z un chiffre ou un trait de soulignement, le test ne fonctionne pas pour d'autres caractères, p. ex. lettre cyrillique ou idéogramme.

Les expressions régulières:

Echappement de / et \

256

```
//Admettons que nous voulons chercher un point.  
//Pour utiliser un caractère spécial en tant que caractère normal,  
//on le précède d'un backslash : \..  
//On appelle aussi cela "échapper un caractère".  
//Par exemple :  
console.log( "Chapter 5.1".match(/\d\.\d/) ); [ '5.1', index: 8, input: 'Chapter 5.1'  
console.log( "Chapter 511".match(/\d\.\d/) ); null  
  
//Les parenthèses sont aussi des caractères spéciaux,  
/* L'exemple ci-dessous recherche une chaîne de caractères "g()":  
console.log( "function g()".match(/g\(\)/) ); [ 'g()', index: 9, input: 'function g()' ]  
  
//Si nous recherchons un backslash \, comme c'est un caractère spécial  
// nous devons donc le doubler.  
console.log( "1\\2".match(/\\"/) ); // '\' [ '\\', index: 1, input: '1\\2', groups: undefined ]
```

```
//Un slash '/' n'est pas un caractère spécial, mais en javascript,  
/* il est utilisé pour ouvrir et fermer l'expression régulière : /...pattern.../,  
nous devons donc aussi l'échapper.  
//?Voici à quoi ressemble une recherche d'un slash '/' :  
console.log( "/".match(/\//) ); [ '/', index: 0, input: '/', groups: undefined ]  
//Par contre si nous n'utilisons pas l'écriture /.../,  
// mais créons l'expression régulière avec new RegExp,  
// alors nous n'avons plus besoin de l'échapper :  
console.log( "/".match(new RegExp("/")) ); [ '/', index: 0, input: '/', groups: undefined ]
```

Les expressions régulières:

Ensembles et intervalles [...]

Plusieurs caractères ou classes de caractères, entourés de crochets [...] signifient "chercher un caractère parmi ceux-là".

Par exemple, [eao] signifie un caractère qui est soit 'a', 'e', ou 'o'.

On appelle cela un *ensemble*. Les ensembles peuvent être combinés avec d'autres caractères dans une même expression régulière

```
// !trouve [t ou m], puis "op"
console.log( "Mop top".match(/[^tm]op/gi) );    [ 'Mop', 'top' ]

console.log( "dd1 dd2".match(/dd[12]/gi) );    [ 'dd1', 'dd2' ]
console.log( "Voila".match(/V[oi]la/) );    null
```

Les crochets peuvent aussi contenir des *intervalles de caractères*.

Par exemple : [a-z] est un caractère pouvant aller de a à z, et [0-5] est un chiffre allant de 0 à 5.

```
// chercher "x" suivi par deux chiffres ou lettres de A à F:
console.log( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) );    [ 'xAF' ]
console.log( "Exception 0xFA".match(/x[0-9A-F][0-9A-F]/g) );    [ 'xFA' ]
```

- \d – équivaut à [0-9],
- \w – équivaut à [a-zA-Z0-9_],
- \s – équivaut à [\t\n\v\f\r], plus quelques autres rares caractères unicodes d'espacement.

Les expressions régulières:

Ensembles et intervalles [...]

258

En plus des intervalles classiques, il existe des intervalles d'exclusion de la forme [^...].

Ils se distinguent par un premier accent circonflexe ^ et correspondent à n'importe quel caractère *à l'exception de ceux contenus dans ces crochets*.

Par exemple :

- [^aeyo] – n'importe quel caractère sauf 'a', 'e', 'y' ou 'o'.
- [^0-9] – n'importe quel caractère à l'exception des chiffres, équivalent à \D.
- [^\s] – tout caractère qui n'est pas un espace, équivalent à \S.

```
// cherche n'importe quel caractère n'étant pas une lettre, un chiffre ou un espace
console.log( "dev10@gmail.com".match(/[^d\sA-Z]/gi) );  [ '@', '.' ]
```

```
//? l'expression régulière [-().^+] cherche un des caractères -().^+:
let c = /[-().^+]/g;
// Pas besoin d'échapper
console.log( "1 + 2 - 3".match(c) );  [ '+', '-' ]
// Si vous décidez de les échapper
// il n'y aura de toute façon aucun d'impact
| c = /\-\(\)\.\.^+/g;
console.log( "1 + 2 - 3".match(c) );  [ '+', '-' ]
```

Les expressions régulières:

Quantificateurs :{n}

Considérons que nous avons une chaîne de caractères +7(903)-123-45-67 et que nous voulons trouver tous les nombres dedans. Mais contrairement à avant nous ne voulons pas seulement trouver les chiffres mais les nombres en entier: 7, 903, 123, 45, 67.

Un nombre est une séquence de 1 ou plus chiffres `\d`. Pour marquer la quantité dont nous avons besoin, nous pouvons ajouter un *quantificateur*.

Le **quantificateur** le plus simple est un nombre entre accolades: `{n}`.

Un quantificateur est attaché à un caractère (ou une classe de caractère, ou un jeu [...], etc) et spécifie la quantité dont nous avons besoin.

```
//! \d{5} indique exactement 5 chiffres, identique à \d\d\d\d\d.
//L'exemple ci-dessous recherche un nombre à 5 chiffres:
console.log("I'm 12345 years old".match(/\d{5}/)); // [ '12345', index: 4, input: 'I'm 12345 years old' ]
//Nous pouvons ajouter \b pour exclure les nombres plus longs: \b\d{5}\b.
```

```
//! La portée: {3,5}, correspond de 3 à 5 fois
//Pour trouver les nombres avec de 3 à 5 chiffres nous pouvons
//mettre les limites entre accolades: \d{3,5}
```

```
console.log("I'm not 12, but 1234 years old".match(/\d{3,5}/)); // [ '1234', index: 16, input: 'I'm not 12, but 1234 years old' ]
//Nous pouvons retirer la limite haute.
/* \d{3,} cherche une séquence de chiffres d'une longueur de 3 ou plus:
console.log("I'm not 12, but 345678 years old".match(/\d{3,}/)); // [ '345678', index: 16, input: 'I'm not 12, but 345678 years old' ]
```

```
let tel = "+7(903)-123-45-67";
let numbers = tel.match(/\d{1,}/g);
console.log(numbers); // [ '7', '903', '123', '45', '67' ]
```

Les expressions régulières:

Quantificateurs :+, *, ?

260

Il y a des abréviations pour les quantificateur les plus utilisés:

+ :signifie “un ou plus”, identique à {1, }.

Par exemple, **\d+** cherche les nombres:

```
let tel = "+7(903)-123-45-67";
let numbers = tel.match(/\d{1,}/g);
console.log(numbers); [ '7', '903', '123', '45', '67' ]
```

?:Signifie “zéro ou plus”, identique à {0,1}. En d’autres termes, il rend le symbole optionnel.

Par exemple, le pattern **o?r** cherche o suivi de zéro ou un u, puis r.

Donc, **colou?r** trouve **color** et **colour**:

```
let color = "Devrais-je écrire color ou colour?";
console.log( color.match(/colou?r/g) ); [ 'color', 'colour' ]
```

*****:Signifie “zéro ou plus”, identique à {0, }. C'est-à-dire que le caractère peut être répété n'importe quel nombre de fois ou bien être absent.

Par exemple, **\d0*** cherche un chiffre suivi de n'importe quel nombre de zéros (plusieurs ou aucun):

```
console.log( "100 10 1".match(/\d0*/g) ); [ '100', '10', '1' ]
//Comparé à + (un ou plus):
console.log( "100 10 1".match(/\d0+/g) ); [ '100', '10' ]
// 1 n'est pas trouvé, puisque 0+ nécessite au moins un zéro
```

Les expressions régulières:

Quantificateurs :+, *, ?

```
//todo :plus de exemples :  
  
//un nombre à virgule flotante): \d+\.\d+  
console.log( "0 1 12.345 7890".match(/\\d+\\.\\d+/g) );    [ '12.345' ]  
// une “balise HTML d’ouverture sans attributs”  
console.log( "<body> ... </body>".match(/<[a-z]+>/gi) );    [ '<body>' ]  
//Cette regexp cherche le caractère '<'  
//suivi par une ou plusieurs lettres Latin, puis '>'.  
  
/*Amélioré:  
console.log( "<h1>Hi!</h1>".match(/<[a-z][a-z0-9]*>/gi) );    [ '<h1>' ]  
  
//“balise HTML d’ouverture ou de fermeture sans attributs”:  
//      /<\/?[a-z][a-z0-9]*>/i  
//Nous avons ajouté un slash optionnel /? près du début du pattern.  
// Nous avons dû l’échapper avec un backslash,  
//sinon Javascript aurait pensé que c’était la fin du pattern.  
console.log( "<h1>Hi!</h1>".match(/<\/?[a-z][a-z0-9]*>/gi) );    [ '<h1>', '</h1>' ]
```

Les expressions régulières:

Groupes ()

262

Une partie de motif peut être entourée de parenthèses (...). Cela s'appelle un "groupe capturant".

Ceci a deux effets :

- 1.Cela permet d'obtenir cette partie de correspondance comme élément du tableau de résultat.
- 2.Si nous mettons après les parenthèses un quantificateur, celui-ci s'applique à tout l'ensemble entre parenthèses.

Sans parenthèses, le motif **go+** signifie le caractère g, suivi par o répété une ou plusieurs fois.

Par exemple, **ooooo** ou **oooooooooo**.

Avec des parenthèses regroupant les caractères, **(go)+** signifie alors **go**, **gogo**, **gogogo** et ainsi de suite.

```
console.log( 'go Gogogo! '.match(/(go)+/ig) ); [ 'go', 'Gogogo' ]
```

Exemples:

```
console.log( "site.com my-site.com my.site.com".match( /([\w-]+\.\.)+\w+/g)); [ 'site.com', 'my-site.com', 'my.site.com' ]
console.log("myEmail@gmail.com name@site.ma".match(/[-.\w]+@[ \w-]+\.\.)+[\w-]+/g)); [ 'myEmail@gmail.com', 'name@site.ma' ]
```

La méthode **str.match(regexp)**, si regexp n'a pas de marqueur g, cherche la première correspondance et la retourne dans un tableau :À l'index 0: la correspondance complète. A l'index 1: le contenu des premières parenthèses. A l'index 2: le contenu des secondes parenthèses etc...

```
let balise = '<h1>Hello, world!</h1>';
let tg = balise.match(/<(.*?)>/);
console.log( tg[0] ); <h1>
console.log( tg[1] ); h1
```

Les expressions régulières:

Groupes ()

Les parenthèses peuvent être **imbriquées**. Dans ce cas la numérotation se fait aussi de gauche à droite.

Par exemple, en effectuant une recherche dans la balise `` nous pourrions être intéressé par :

- 1.Son contenu complet : `span class="my"`.
- 2.Son nom : `span`.
- 3.Ses attributs : `class="my"`.

```



```

`<(([a-z]+)\s*([^>]*))>`

```

let span = '<span class="my">';
let sp = /<(([a-z]+)\s*([^>]*))>/;

let resultat = span.match(sp);
console.log(resultat[0]);  <span class="my">
console.log(resultat[1]);  span class="my"
console.log(resultat[2]);  span
console.log(resultat[3]);  class="my"
  
```

un groupe est optionnel s'il a le quantificateur `(...)?` son élément correspondant est présent dans le tableau est vaut `undefined`.

Par exemple: considérons l'expression régulière `a(z)?(c)?.`

Cela cherche un "a" suivi d'un éventuel "z" suivi d'un éventuel "c".

Si nous lançons une recherche sur la seule lettre a, alors le résultat donne:

```

let match = 'a'.match(/a(z)?(c)?./);
let match1 = 'ac'.match(/a(z)?(c)?./)

console.log( match.length );  3
console.log( match[0] );  a
console.log( match1[0] );  ac
console.log( match[1] );  undefined
console.log( match[2] );  undefined
console.log( match1[2] );  c
  
```

Les expressions régulières: groupes avec matchAll

264

matchAll est une méthode récente, La méthode matchAll n'est pas supportée par d'anciens navigateurs.

Tout comme match, elle cherche des correspondances, mais avec 3 différences :

- 1.Elle ne retourne pas de tableau, mais un itérateur.
- 2.Si le marqueur g est présent, elle retourne toutes les correspondances dans des tableaux avec les groupes.
- 3.S'il n'y a pas de correspondance, elle ne retourne pas null, mais un itérateur vide.

```
let results = '<h1> <h2>'.matchAll(/<(.*)>/gi);
// results - n'est pas un tableau, mais un itérateur
console.log(results);  { [Iterator] }
console.log(results[0]);  undefined
results = Array.from(results); //convertissons-le en tableau
console.log(results[0]);  [ '<h1>', 'h1', index: 0, input: '<h1> <h2>', groups: undefined ]
console.log(results[1]);  [ '<h2>', 'h2', index: 5, input: '<h1> <h2>', groups: undefined ]

//Il n'y a pas besoin de Array.from si nous bouclons sur le résultat :
for(let result of results) {
  console.log(result);  ... , 'h1', index: 0, input: '<h1> <h2>', groups: undefined ], [ '<h2>', 'h2', index: 5,
}
//Ou bien en déstructurant :
💡
let [tag1, tag2] = results;

console.log( tag1[0] );  <h1>
console.log( tag1[1] );  h1
console.log( tag1.index );  0
console.log( tag1.input );  <h1> <h2>
```

Les expressions régulières: groupes nommés

265

Il est difficile de se souvenir de groupes par leur numéro. Bien que faisable pour des motifs simples, cela devient ardu dans des motifs plus complexes. Il existe une meilleure option : **nommer les parenthèses**.

Cela se fait en mettant ?<name> immédiatement après la parenthèse ouvrante.

Par exemple, recherchons une date au format “year-month-day”:

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let date = "2019-04-30";
//!les groupes figurent dans la propriété .groups de la correspondance.
let groups = date.match(dateRegexp).groups;

console.log(groups.year);    2019
console.log(groups.month);   04
console.log(groups.day);     30

/*Pour chercher toutes les dates, nous pouvons ajouter le marqueur g.
/* matchAll pour obtenir des correspondances complètes, avec les groupes :
let dateRegexp2 = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let dates = "2019-10-30 2020-01-01";

let allDates = dates.matchAll(dateRegexp2);

for(let result of allDates) {
  let {year, month, day} = result.groups;

  console.log(` ${day}.${month}.${year}`); 30.10.2019, 01.01.2020
}
```

Les expressions régulières:

Groupes capturant dans un remplacement

La méthode `str.replace(regexp, replacement)` qui remplace dans `str` toutes les correspondances de `regexp`, nous permet d'utiliser le contenu des parenthèses dans la chaîne de `replacement`. Nous utiliserons alors `$n`, où `n` correspond au numéro de groupe.

Pour les parenthèses nommées la référence au groupe se fera avec `$<name>`.

Par exemple, reformatons les dates depuis le format “year-month-day” vers “day.month.year”:

```
let fullName = "Adam Mohamed";
let rp = /(\w+) (\w+)/;

console.log( fullName.replace(rp, '$2, $1') ); | Mohamed, Adam
```

```
let d = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;
dates = "2022-10-30, 2023-01-23";
console.log( dates.replace(d, '$<day>.$<month>.$<year>') ); | 30.10.2022, 23.01.2023
```

Nous avons parfois besoin de parenthèses pour appliquer correctement un quantificateur, sans avoir besoin de leurs contenu dans les résultats.
Un groupe peut être exclu des résultats en ajoutant `?:` au début.
Par exemple, si nous voulons trouver `(dev)+`, sans avoir les contenus des parenthèses `(dev)` comme élément du tableau de correspondance, nous pouvons écrire : `(?:dev)+`.

Dans l'exemple suivant nous obtenons seulement `digital` comme élément supplémentaire de la correspondance.:

```
let filiere = "devdev digital!";

// ?: exclu 'dev' d'une capture
let f = /(?:dev)+ (\w+)/i;

let r = filiere.match(f);

console.log( r[0] ); | devdev digital
console.log( r[1] ); | digital
console.log( r.length ); | 2
```

Les expressions régulières:

Alternance (OU): |

267

Alternance est le terme d'expression régulière qui représente un "OU".

Dans une expression régulière l'alternance est représentée par une barre verticale |.

Par exemple, nous souhaitons trouver les langages de programmation suivants: HTML, PHP, Java ou JavaScript.

La regexp correspondante : `html|php|java(script)?`

```
let lang = /html|php|css|java(script)?/gi;
let langugaes = "D'abord HTML est apparu, puis CSS, puis JavaScript";
console.log( langugaes.match(lang) ); // [ 'HTML', 'CSS', 'JavaScript' ]
```

les crochets permettent de choisir entre plusieurs caractères, par exemple `gr[ae]y` correspond à `gray` ou `grey`.

Les crochets n'autorisent que les caractères ou les classes de caractère.

L'alternance permet n'importe quelle expression. Une

regexp `A|B|C` signifie A, B ou C.

Par exemple:

- `gr(a|e)y` signifie la même chose que `gr[ae]y`.

- `gra|ey` signifie `gra` ou `ey`.

Pour appliquer l'alternance à une partie du modèle nous pouvons l'encadrer entre parenthèses:

- `I love HTML|CSS` correspond à `I love HTML` ou `CSS`.

- `I love (HTML|CSS)` correspond à `I love HTML` ou `I love CSS`.

Exemple :

construire une regexp pour trouver un temps de la forme hh:mm

```
let time = /([01]\d|2[0-3]):[0-5]\d/g;
console.log("00:00 10:10 23:59 25:99 1:2" .match(time));
```

Les expressions régulières: lookahead” et “lookbehind”

Parfois nous avons juste besoin de trouver les motifs précédents ou suivant un autre motif.

Il existe pour cela des syntaxes spéciales, appelées “**lookahead**” et “**lookbehind**”, ensemble désignées par “**lookaround**”.

Exemples :

```
let dinde = "1 Dinde Coûte $30";
console.log( dinde.match(/\d+(?=$)/) );   [ '30', index: 15, input: '1 Dinde Coûte $30' ]
// le nombre 1 est ignoré
// vu qu'il n'est pas suivi de €

console.log( dinde.match(/\d+(?=\.s)(?=.*30)/) );   [ '1', index: 15, input: '1 Dinde Coûte $30' ]
// le signe dollar est échappé \$
console.log( dinde.match(/(?<=\$)\d+/) );   [ '30', index: 15, input: '1 Dinde Coûte $30' ]
// (ignore le nombre sans dinde)

let turkey = "2 dindes coûtent $60";
console.log( turkey.match(/\d+\b(?!\$/g) ) );   [ '2' ]
//(le prix ne correspond pas au motif)

console.log( turkey.match(/(?<!\$)\b\d+/g) );   [ '2' ]
// 2 (le prix ne correspond pas )

//groupes
let dollar = /\d+(?=(dh|\$))/; // parenthèses supplémentaires au début
console.log( turkey.match(dollar) );   [ '60', '', index: 18, input: '2 dindes coûtent $60' ]
```

motif	type	correspondances
X(?:=Y)	Lookahead positif	X si il est suivi de Y
X(?:!=Y)	Lookahead négatif	X si il n'est pas suivi de Y
(?<=Y)X	Lookbehind positif	X s'il suit Y
(?<!=Y)X	Lookbehind négatif	X s'il ne suit pas Y

Validation du Form en utilisant expreg: Exemple

Inscription

Nom utilisateur :

Email :

inserer un email valide

Password

Password devrait au moins 8 caractères.

Password à nouveau :

passwords ne sont pas équivalents

S'inscrire

Inscription

Nom utilisateur :

Email :

Password

Password à nouveau :

S'inscrire

Validation du Form en utilisant expreg:

Exemple

270

Index.html

```

<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Poppins:wght@400;700&display=swap" rel="stylesheet">
<link rel="stylesheet" href="./style.css">
<script defer src="./index.js"></script>

</head>
<body>
    <div class="container">
        <form id="form" action="/">
            <h1>Inscription</h1>
            <div class="input-control">
                <label for="username">Nom utilisateur :</label>
                <input id="username" name="username" type="text">
                <div class="error"></div>
            </div>
            <div class="input-control">
                <label for="email">Email :</label>
                <input id="email" name="email" type="text">
                <div class="error"></div>
            </div>
            <div class="input-control">
                <label for="password">Password</label>
                <input id="password" name="password" type="password">
                <div class="error"></div>
            </div>
        </form>
    </div>
</body>
</html>

```

Validation du Form en utilisant expreg: Exemple

Style.css

```
body {background: linear-gradient(to right, #0f2027, #203a43, #2c5364);font-family: 'Poppins', sans-serif;}
```

```
#form {width: 300px; margin: 20vh auto 0 auto; padding: 20px; background-color: #whitesmoke; border-radius: 4px; font-size: 12px;}
```

```
#form h1 {color: #0f2027; text-align: center;}
```

```
#form button {padding: 10px; margin-top: 10px; width: 100%; color: #white; background-color: #rgb(41, 57, 194); border: none; border-radius: 4px;}
```

```
.input-control { display: flex; flex-direction: column;}
```

```
.input-control input { border: 2px solid #f0f0f0; border-radius: 4px; display: block; font-size: 12px; padding: 10px; width: 100%;}
```

```
.input-control input:focus { outline: 0;}
```

```
.input-control.success input { border-color: #09c372;}
```

```
.input-control.error input { border-color: #ff3860;}
```

```
.input-control .error { color: #ff3860; font-size: 9px; height: 13px;}
```

Validation du Form en utilisant expreg: Exemple

```
const form = document.getElementById('form');
const username = document.getElementById('username');
const email = document.getElementById('email');
const password = document.getElementById('password');
const password2 = document.getElementById('password2');

form.addEventListener('submit', e => {
    e.preventDefault();
    validateInputs();
});

const setError = (element, message) => {
    const inputControl = element.parentElement;
    const errorDisplay = inputControl.querySelector('.error');
    errorDisplay.innerText = message;
    inputControl.classList.add('error');
    inputControl.classList.remove('success')
}
```

Validation du Form en utilisant expreg

Exemple:

273

Index.js

```
const setSuccess = element => {
  const inputControl = element.parentElement;
  const errorDisplay = inputControl.querySelector('.error');

  errorDisplay.innerText = '';
  inputControl.classList.add('success');
  inputControl.classList.remove('error');
};

const isValidEmail = email => {
  const re = /^(([^\<>()[]\\.,;:\\s@"]+(\.[^\<>()[]\\.,;:\\s@"]+)*|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}|\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.\[[a-zA-Z\-\_0-9]+\.\.)+[a-zA-Z]{2,}))$/;
  return re.test(String(email).toLowerCase());
}
```

Expression régulière complète pour tester l'email

```
const re = /^(([^\<>()[]\\.,;:\\s@"]+(\.[^\<>()[]\\.,;:\\s@"]+)*|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}|\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.\[[a-zA-Z\-\_0-9]+\.\.)+[a-zA-Z]{2,}))$/;
```

Validation du Form en utilisant expreg:

Exemple

274

Index.js

```

const validateInputs = () => {
  const usernameValue = username.value.trim();
  const emailValue = email.value.trim();
  const passwordValue = password.value.trim();
  const password2Value = password2.value.trim();

  if(usernameValue === '') {
    setError(username, 'nom utilisateur est obligatoire');
  } else {
    setSuccess(username);
  }

  if(emailValue === '') {
    setError(email, 'Email est obligatoire');
  } else if (!isValidEmail(emailValue)) {
    setError(email, 'inserer un email valide');
  } else {
    setSuccess(email);
  }

  if(passwordValue === '') {
    setError(password, 'Password est obligatoire');
  } else if (passwordValue.length < 8 ) {
    setError(password, 'Password devrait au moins 8 caractères.');
  } else {
    setSuccess(password);
  }

  if(password2Value === '') {
    setError(password2, 'confirme votre password');
  } else if (password2Value !== passwordValue) {
    setError(password2, "passwords ne sont pas équivalents");
  } else {
    setSuccess(password2);
  }
};


```

EP39 :

275

Quelle chaîne de caractères correspond à ^\$?

complétez le code pour qu'il avoir
le résultats suivant :

```
console.log( "JavaScript".match(/[^cript]/) ); [ 'JavaS'
```

1.L'heure à un format : **hours:minutes**. Les heures et les minutes ont deux chiffres, comme **09:00**.

Écrire une expression régulière pour trouver l'heure dans la chaîne de caractère : début cours à 08:30 dans la salle 06.

2. Écrire une expression rationnelle pour trouver l'heure quelle que soit sa forme :

```
console.log( "début cours à 08:30. fin à 13-30".match( ) ); [ '08:30', '13-30' ]
```

Créer une regexp pour trouver une ellipse:
3 (ou plus?) points à la suite.

```
let rgp =  
  console.log( "bonjour!... etc.....".match(rgp) ); [ '...', '.....' ]
```

Créez une regexp pour trouver les couleurs HTML écrites comme #ABCDEF:
d'abord # puis 6 caractères hexadécimaux.

```
let clr=  
// une couleur  
console.log( "#123456".match( clr ) ); [ '#123456' ]  
// pas une couleur  
console.log( "#12345678".match( clr ) ); null
```

Créer une regexp pour trouver les mots en
doubles guillemets dans une phrase

```
let gui =  
let phrase = 'le "PHP" et "Java" ';  
console.log( phrase.match(gui) ); [ '"PHP"', '"Java"' ]
```

EP40:

276

Trouvez tous les commentaires HTML dans un texte :

```
let cmt =  
  
let comments = `... <!-- comment 1  
| test --> .. <!----> ..  
`;  
console.log( comments.match(cmt) ); [ '<!-- comment 1\n test -->', '<!---->' ]
```

Écrire une expression régulière pour trouver toutes les balises HTML (ouvriantes et fermantes) avec leurs attributs.

```
let tag =  
let balises = '<> <a href="/"> <input type="radio" checked> <b>';  
console.log( balises.match(tag) ); [ '<a href="/">', '<input type="radio" checked>', '<b>' ]
```

L'adresse MAC d'une interface réseau est constitué de 6 paires de nombres hexadécimaux séparées par un double point.

Par exemple : '01:32:54:67:89:AB'.

Écrire une regexp qui vérifie qu'une chaîne de caractères soit bien une adresse MAC.

```
('01:32:54:67:89:AB') ); true  
('0132546789AB') ); false  
('01:32:54:67:89') ); false  
('01:32:54:67:89:ZZ') ) false
```

Écrire une RegExp qui correspond à des couleurs au format #abc ou #abcdef. C'est à dire : # suivi par 3 ou 6 chiffres hexadécimaux.

```
let couleur =  
let colors = "color: #3f3; background-color: #AA00ef;: #abcd";  
console.log( colors.match(couleur) ); [ '#3f3', '#AA00ef', '#abc' ]
```

EP41:

277

1. Écrire un regexp qui cherche tous les nombres décimaux, comprenant les entiers, les nombres décimaux avec le point comme séparateur et les nombres négatifs.

```
let rg =
let nombres = "-1.5 0 2 -123.4 - .";
console.log( nombres.match(rg) ); [ '-1.5', '0', '2', '-123.4' ]
```

2. Créez une expression régulière qui ne recherche que les expressions non négatives (zéro est autorisé)

```
let positif =
console.log( nombres.match(positif) ); [ '5', '0', '2', '4' ]
```

Une expression arithmétique consiste en 2 nombres et un opérateur entre les deux.

Créez une fonction **parse(expr)** qui prend une expression et retourne un tableau de trois éléments sans espaces :

1. Le premier nombre.
2. L'opérateur.
3. Le second nombre.

```
console.log( parse("-1.23 * 3.45") ); [ '-1.23', '*', '3.45', ]
```

Créez une regexp qui trouve les langages de programmation dans une chaîne de caractères: Java JavaScript PHP C++ C

```
let lg =
let prog = "Java, JavaScript, PHP, C, C++";
console.log( prog.match(lg) ); [ 'Java', 'JavaScript', 'PHP', 'C', 'C++' ]
```

Écrivez une regexp pour trouver la balise `<style...>`.

Il devrait trouver la balise en entier: il pourrait ne pas avoir d'attributs `<style>` ou en avoir plusieurs `<style type="..." id="...">`.... Mais la regexp ne devrait pas trouver `<styler>`!

```
let style =
console.log( '<style> <styler> <style test="...">'.match(style) ); [ '<style>', '<style test="...">' ]
```

TP9:

- ▶ Soit les liens suivants ,déterminer une expreg pour déterminer les liens valides
- ▶ Parcourir les liens et afficher les liens valides !

```
<li><a href="https://www.google.com">Google</a></li>
<li><a href="https://www.facebook.com">Facebook</a></li>
<li><a href="https://www.twitter.com">Twitter</a></li>
<li><a href="https://www.linkedin.com">LinkedIn</a></li>
<li><a href="mailto:contact@example.com">Nous contacter</a></li>
```

Vérifier les champs du formulaire respectent les critères suivants :

- Le champ "Nom" doit contenir uniquement des lettres (majuscules ou minuscules) et des espaces.
- Le champ "Email" doit respecter le format d'une adresse email valide.
- Le champ "Mot de passe" doit contenir au moins 8 caractères, avec au moins une lettre majuscule, une lettre minuscule et un chiffre.

Nom :

Email :

Mot de passe :

Envoyer

Partie 10 :

Orienté Objet

- objets: getters & setters
- Héritage prototypal
- prototype
- classes
- Accesseurs/Mutateurs
- Héritage
- surcharge
- méthodes statiques
- mixins
- MVC

Objets :getters & setters

280

Les propriétés d'accès sont représentées par les méthodes "getter" et "setter". Dans un littéral d'objet, ils sont notés get et set :

```
let obj = {
  get propName() {
    // getter, the code executed on getting obj.propName
  },
  set propName(value) {
    // setter, the code executed on setting obj.propName = value
  }
};
```

Maintenant, nous voulons ajouter une propriété fullName, qui devrait être "John Smith". Bien sûr, nous ne voulons pas copier-coller des informations existantes, nous pouvons donc les implémenter en tant qu'accesseurs :

```
let user = {
  name: "John",
  surname: "Smith",
  get fullName() {
    return `${this.name} ${this.surname}`;
  }
};

alert(user.fullName); // John Smith
```

fullName n'a qu'un getter. Si nous essayons d'attribuer user.fullName=, il y aura une erreur :

```
set fullName(value) {
  [this.name, this.surname] = value.split(" ");
}

// set fullName is executed with the given value.
user.fullName = "Alice Cooper";

alert(user.name); // Alice
alert(user.surname); // Cooper
```

Getters & setters

mohamed@goumih.com

281

si nous voulons interdire les noms trop courts pour l'utilisateur, nous pouvons avoir un nom de setter et conserver la valeur dans une propriété séparée `_name` :

```
function User(name, age) {  
    this.name = name;  
    this.age = age;  
}  
  
let john = new User("John", 25);  
  
alert( john.age ); // 25
```

Mais tôt ou tard, les choses peuvent changer. Au lieu de l'âge, nous pouvons décider de stocker l'anniversaire, car c'est plus précis et plus pratique :

```
function User(name, birthday) {  
    this.name = name;  
    this.birthday = birthday;  
}  
  
let john = new User("John", new Date(1992, 6, 1));
```

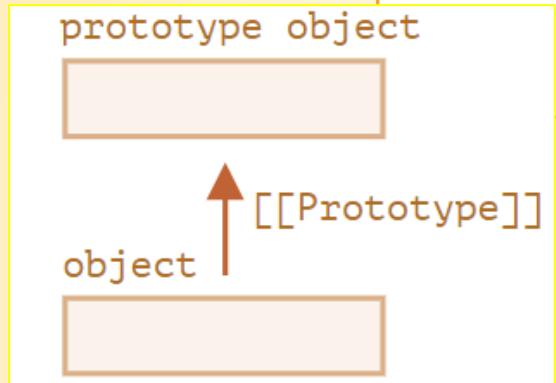
```
let user = {  
    get name() {  
        return this._name;  
    },  
  
    set name(value) {  
        if (value.length < 4) {  
            alert("Name is too short, need at least 4 characters");  
            return;  
        }  
        this._name = value;  
    }  
};  
  
user.name = "Pete";  
alert(user.name); // Pete  
  
user.name = ""; // Name is too short...
```

```
let john = new User("John", new Date(1992, 6, 1));  
  
alert( john.birthday ); // birthday is available  
alert( john.age ); // ...as well as the age
```

Héritage prototypal

282

En JavaScript, les objets ont une propriété cachée spéciale `[[Prototype]]` qui est soit `null` ou fait référence à un autre objet. Cet objet s'appelle "un prototype" :



Lorsque nous lisons une propriété depuis `object`, et qu'elle est manquante, JavaScript la prend automatiquement du prototype. En programmation, une telle chose est appelée "héritage prototypal".

La propriété `[[Prototype]]` est interne et cachée, mais il y a plusieurs façons de la définir.
L'un d'eux est d'utiliser le nom spécial `proto`, comme ceci :

```

let animal = {
  eats: true
};

let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal; // sets rabbit.[ [ Prototype ] ] = animal
  
```

Ici, nous pouvons dire que "animal est le prototype de rabbit" ou que "rabbit hérite de manière prototypal de animal".

Donc, si `animal` a beaucoup de propriétés et de méthodes utiles, elles deviennent automatiquement disponibles dans `rabbit`. De telles propriétés sont appelées "héritées".

Si nous recherchons une propriété dans `rabbit`, et qu'elle en manque, JavaScript la prend automatiquement à partir de `animal`.

```

// nous pouvons maintenant trouver les deux propriétés dans rabbit:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true
  
```

Héritage prototypal

283

Si nous avons une méthode dans `animal`, elle peut être appelée sur `rabbit` :
 La chaîne de prototypes peut être plus longue

```
let animal = {
  eats: true,
  walk() {
    /* cette méthode ne sera pas utilisée par rabbit */
  }
};

let rabbit = {
  __proto__: animal
};

rabbit.walk = function() {
  alert("Rabbit! Bounce-bounce!");
};

rabbit.walk(); // Rabbit! Bounce-bounce!
```

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

// walk est prise à partir du prototype
rabbit.walk(); // Animal walk
```

```
let animal = {
  eats: true,
  walk() {
    alert("Animal walk");
  }
};

let rabbit = {
  jumps: true,
  __proto__: animal
};

let longEar = {
  earLength: 10,
  __proto__: rabbit
};

// walk est prise à partir de la chaîne de prototype
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (de rabbit)
```

Héritage prototypal

mohamed@goumih.com

284

```
let user = {
    name: "John",
    surname: "Smith",

    set fullName(value) {
        [this.name, this.surname] = value.split(" ");
    },

    get fullName() {
        return `${this.name} ${this.surname}`;
    }
};

let admin = {
    __proto__: user,
    isAdmin: true
};

alert(admin.fullName); // John Smith (*)

// le mutateur se déclanche!
admin.fullName = "Alice Cooper"; // (**)

alert(admin.fullName); // Alice Cooper, state of ...
alert(user.fullName); // John Smith, state of user
```

ici dans la ligne (*) la propriété `admin.fullName` a un accesseur dans le prototype `user`, donc c'est appelé.

Et dans la ligne (**) la propriété a un mutateur dans le prototype, donc c'est appelé.

```
let animal = {
    eats: true
};

let rabbit = {
    jumps: true,
    __proto__: animal
};
```

```
// Object.keys ne renvoie que ses propres clés
alert(Object.keys(rabbit)); // jumps
```

```
// for..in boucle sur les clés propres et héritées
for(let prop in rabbit) alert(prop); // jumps, puis eats
```

La boucle for...in

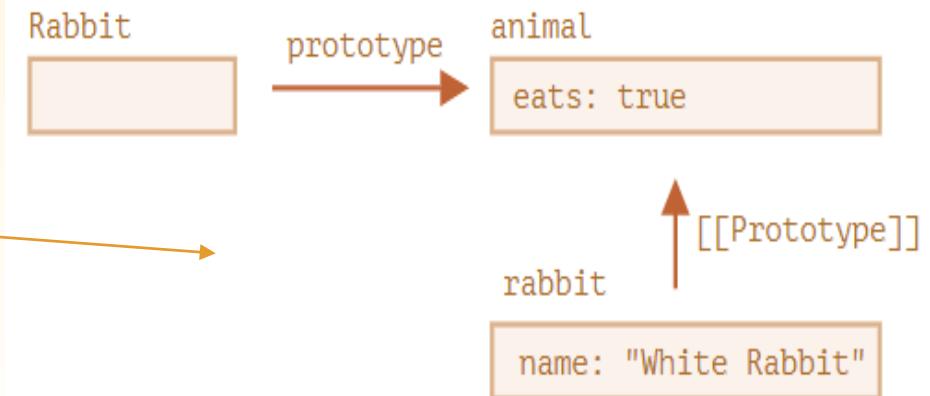
La boucle `for...in` itère aussi sur les propriétés héritées.

Prototype

mohamed@goumih.com

285

Définir `Rabbit.prototype=animal` énonce littéralement ce qui suit: "Lorsqu'un new Rabbit est créé, assigner son [[Prototype]] à animal".



```
let animal = {  
    eats: true  
};  
  
function Rabbit(name) {  
    this.name = name;  
}  
  
Rabbit.prototype = animal;  
  
let rabbit = new Rabbit("White Rabbit"); // rabbit.__proto__ == animal  
  
alert( rabbit.eats ); // true
```

Classes

286

La syntaxe de base est:

```
class MyClass {
    // Les méthodes de la classe
    constructor() { ... }
    method1() { ... }
    method2() { ... }
    method3() { ... }
    ...
}
```

Vous pouvez ensuite utiliser `new MyClass()` pour créer un nouvel objet ayant toute la liste des méthodes.

La méthode `constructor()` est automatiquement appelée par `new`, donc nous pouvons initialiser l'objet à ce niveau.

Tout comme les fonctions, les classes peuvent être définies à l'intérieur d'une autre expression, passées en paramètres, retournées, assignées etc.

Voici un exemple d'expression d'une classe:

```
let User = class {
    sayHi() {
        alert("Hello");
    }
};
```

Nous pouvons même créer les classes dynamiquement "à la demande", comme ainsi :

Dans le JavaScript moderne, il y a une construction de la "classe" plus avancée qui introduit de nombreux nouveaux aspects utiles en langage orienté objet.

```
class User {
    constructor(name) {
        this.name = name;
    }

    sayHi() {
        alert(this.name);
    }
}

// Usage:
let user = new User("John");
user.sayHi();
```

```
function makeClass(phrase) {
    // déclare une classe et la retourne
    return class {
        sayHi() {
            alert(phrase);
        }
    };
}

// Cree une nouvelle classe
let User = makeClass("Hello");

new User().sayHi(); // Hello
```

Accesseurs/Mutateurs

287

Tout comme les objets littéraux, les classes peuvent inclure des accesseurs/mutateurs, des propriétés évaluées etc.

Voici un exemple pour `user.name` implémenté en utilisant les propriétés **get/set**:

```
class User {  
  
    constructor(name) {  
        // invoque l'accesseur (the setter)  
        this.name = name;  
    }  
  
    get name() {  
        return this._name;  
    }  
  
    set name(value) {  
        if (value.length < 4) {  
            alert("Name is too short.");  
            return;  
        }  
        this._name = value;  
    }  
  
}  
  
let user = new User("John");  
alert(user.name); // John  
  
user = new User(""); // le nom est trop court.
```

Héritage de classe

288

L'héritage de classe est un moyen pour une classe d'étendre une autre classe.

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed = speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }
}

let animal = new Animal("My animal");
```

```
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}

let rabbit = new Rabbit("White Rabbit");

rabbit.run(5); // White Rabbit court à la vitesse 5.
rabbit.hide(); // White Rabbit se cache!
```

Ajoutons maintenant un constructeur personnalisé à Rabbit. Il spécifiera le earLength en plus de name

```
constructor(name, earLength) {
  this.speed = 0;
  this.name = name;
  this.earLength = earLength;
}
```

```
stop() {
  super.stop(); // appeler stop du parent
  this.hide(); // puis hide
}
```

- `super.method(...)` pour appeler une méthode parente.
- `super(...)` pour appeler un constructeur parent (dans notre constructeur uniquement).

```
// now fine
let rabbit = new Rabbit("White Rabbit", 10);
alert(rabbit.name); // White Rabbit
alert(rabbit.earLength); // 10
```

Surcharge

289

```
class Animal {  
    showName() { // au lieu de this.name = 'animal'  
        alert('animal');  
    }  
  
    constructor() {  
        this.showName(); // au lieu de alert(this.name);  
    }  
}  
  
class Rabbit extends Animal {  
    showName() {  
        alert('rabbit');  
    }  
}  
  
new Animal(); // animal  
new Rabbit(); // rabbit
```

L'opérateur `instanceof` permet de vérifier si un objet appartient à une certaine classe. Il prend également en compte l'héritage.

Une telle vérification peut être nécessaire dans de nombreux cas. Nous l'utilisons ici pour construire une fonction *polymorphe*, celle qui traite les arguments différemment en fonction de leur type.

```
class Rabbit {}  
let rabbit = new Rabbit();  
  
// est-ce un objet de la classe Rabbit?  
alert( rabbit instanceof Rabbit ); // true
```

```
let arr = [1, 2, 3];  
alert( arr instanceof Array ); // true  
alert( arr instanceof Object ); // true
```

Propriétés et méthodes statiques

mohamed@goumih.com

290

Nous pouvons aussi assigner une méthode à la fonction de classe elle-même, pas à son "prototype". De telles méthodes sont appelées *statique*.

Dans une classe, ils sont précédés du mot clé **static**, comme ceci:

```
class User {  
    static staticMethod() {  
        alert(this === User);  
    }  
}  
  
User.staticMethod(); // true
```

```
class User {}  
  
User.staticMethod = function() {  
    alert(this === User);  
};  
  
User.staticMethod(); // true
```

La valeur de **this** dans l'appel **User.staticMethod()** est le constructeur de la classe **User** lui-même (la règle "objet avant le point").

Généralement, les méthodes statiques sont utilisées pour implémenter des fonctions appartenant à la classe, mais pas à un objet particulier de celle-ci.

Par exemple, nous avons des objets **Article** et avons besoin d'une fonction pour les comparer.

Une solution naturelle serait d'ajouter la méthode **Article.compare**, comme ceci:

```
class Article {  
    constructor(title, date) {  
        this.title = title;  
        this.date = date;  
    }  
  
    static compare(articleA, articleB) {  
        return articleA.date - articleB.date;  
    }  
}
```

```
// usage  
let articles = [  
    new Article("HTML", new Date(2019, 1, 1)),  
    new Article("CSS", new Date(2019, 0, 1)),  
    new Article("JavaScript", new Date(2019, 11, 1))  
];  
  
articles.sort(Article.compare);  
  
alert(articles[0].title); // CSS
```

Les mixins

291

En JavaScript, nous ne pouvons hériter que d'un seul objet. Il ne peut y avoir qu'un [[Prototype]] pour un objet. Et une classe peut étendre qu'une seule autre classe.

un mixin est une classe contenant des méthodes qui peuvent être utilisées par d'autres classes sans avoir à en hériter. En d'autres termes, un ***mixin*** fournit des méthodes qui implémentent un certain comportement, mais nous ne l'utilisons pas seul, nous l'utilisons pour ajouter le comportement à d'autres classes.

Par exemple ici, le mixin `sayHiMixin` est utilisé pour ajouter un peu de "discours" à `User`:

```
// mixin
let sayHiMixin = {
  sayHi() {
    alert(`Hello ${this.name}`);
  },
  sayBye() {
    alert(`Bye ${this.name}`);
  }
};
```

```
// copier les méthodes
Object.assign(User.prototype, sayHiMixin);

// maintenant User peut dire bonjour
new User("Dude").sayHi(); // Hello Dude!
```

```
// usage:
class User {
  constructor(name) {
    this.name = name;
  }
}
```

Il n'y a pas d'héritage, mais une méthode de copie simple. Ainsi, `User` peut hériter d'une autre classe et inclure le mixin pour ajouter les méthodes supplémentaires, comme ceci:

```
class User extends Person {
  // ...
}

Object.assign(User.prototype, sayHiMixin);
```

Créez une fonction constructeur **Person(first, last, age, gender, interests)** qui permet de créer un Person avec les propriétés suivantes :name(un objet de first et last) ,age ,gender ,interests(un tableau des interests).

La function contient une function **bio** qui affiche les données d'un Person elle doit verifier si le Person est femme ou homme ;pour commencer l'affichage par il ou elle .

```
let person1 = new Person('Ahmed', 'Mohamed', 25, 'homme', ['foot', 'lecture']);  
|  
Ahmed Mohamed a 25 ans. Il aime foot et lecture.
```

1. Créez une classe appelée **Person** avec les propriétés **name** et **age**, et une méthode appelée **sayHello()** qui enregistre un message d'accueil dans la console.
2. Créez une classe appelée **Student** qui hérite de la classe Person et ajoute la propriété **studentId**. Remplacez la méthode **sayHello()** pour inclure l'ID étudiant.
3. Ajoutez une méthode statique à la classe Person appelée **getAverageAge()** qui prend un tableau d'objets **Person** et renvoie l'âge moyen des personnes dans le tableau.

Créez une classe appelée **Rectangle** avec les propriétés **width** et **height**, et ajoutez des méthodes **getter** et **setter** pour les deux propriétés qui valident que la valeur est un nombre positif.
Et une méthode qui calcule la surface du rectangle **area()**

EP43:

293

1. Créez une classe **Product** avec les propriétés :**name et price et** une méthode **toString** qui affiche le name et le price en DH.
2. Créer 3 **Products** avec cette classe.
3. Créer un classe **ShoppingCart** avec les propriétés **products**(Tableau vide au début) ,**total_price**(0 au début),**no_discount_price**(0 au début) et **discount**(0 au début).
4. Ajouter dans **ShoppingCart** la méthode **addProduct(Product)** qui permet d'ajouter un Product au tableau products et augmente le total_price par price et affecte total_price au no_discount_price
5. Créer une méthode **quantity_check()** qui vérifié si le client a acheté 5 ou plus Products si oui il diminué le total_price de 10% sinon le no_discount_price.
6. Créer une méthode **product_check()** qui vérifié si le client a acheté 3 item du même Product si oui on le donne un Product free(c-à-dire) il paye seulement pour les 3 items.
7. Créer une méthode **toString()** qui affiche les Products acheté par un client et le total_price à payé.
8. Tester ses méthodes en ajoutant plusieurs Products.

Créez une classe **Person** avec une propriété privée **_name** à l'aide d'un **WeakMap**(utiliser set et get de weakMap)

MVC :Model ,View ,Controller

294

Qu'est-ce que le contrôleur de vue modèle ?

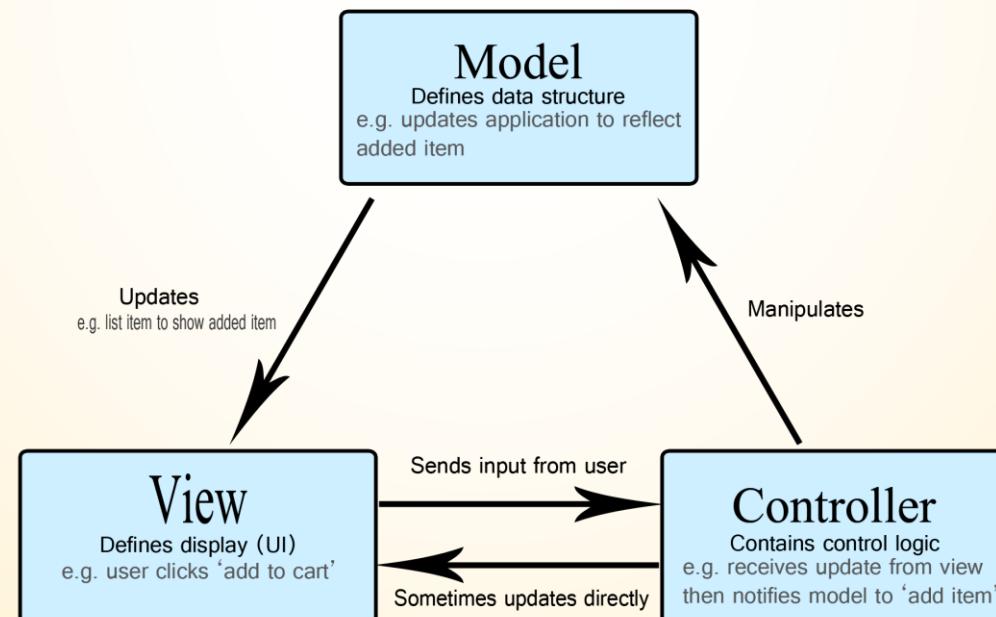
MVC est un modèle possible pour organiser votre code

Modèle : Gère les données d'une application

Vue : Une représentation visuelle du modèle

Contrôleur : relie le modèle et la vue. Il prend l'entrée de l'utilisateur, comme cliquer ou taper, et gère les rappels pour les interactions de l'utilisateur.

Le modèle ne touche jamais la vue. La vue ne touche jamais le modèle. Le contrôleur les relie.



Exemple 1:MVC

295

mvc0.js

```
// Model
class CounterModel {
  constructor() {
    this.count = 0;
  }
  increment() {
    this.count++;
  }
  decrement() {
    this.count--;
  }
  getCount() {
    return this.count;
  }
}
```

```
<div>
  <p>Compteur: <span id="count"></span></p>
  <button id="incrementBtn">+</button>
  <button id="decrementBtn">-</button>
</div>
<script src="mvc0.js"></script>
```

Compteur: 0



Compteur: 5



```
// View
class CounterView {
  constructor() {
    this.countDisplay = document.getElementById('count');
    this.incrementBtn = document.getElementById('incrementBtn');
    this.decrementBtn = document.getElementById('decrementBtn');
    console.log(this.incrementBtn);
  }
  updateCount(count) {
    this.countDisplay.innerText = count;
  }
  bindIncrementButton(handler) {
    this.incrementBtn.addEventListener('click', handler);
  }
  bindDecrementButton(handler) {
    this.decrementBtn.addEventListener('click', handler);
  }
}
```

Exemple 1:MVC

296

mvc0.js (suite)

```
// Controller
class CounterController {
  constructor(model, view) {
    this.model = model;
    this.view = view;
    this.view.bindIncrementButton(this.handleIncrement);
    this.view.bindDecrementButton(this.handleDecrement);
    // Affichage de comptage initial
    this.updateView();
  }
  handleIncrement = () => {
    this.model.increment();
    this.updateView();
  }
  handleDecrement = () => {
    this.model.decrement();
    this.updateView();
  }
  updateView() {
    const count = this.model.getCount();
    this.view.updateCount(count);
  }
}
```

```
// Usage
const model = new CounterModel();
const view = new CounterView();
const controller = new CounterController(model, view);
```

Exemple2 :MVC

297

Dans cette application **Ajouter un Task**, ce sera le HTML rendu dans le DOM et le CSS.
Le contrôleur connecte le modèle et la vue.
Il prend l'entrée de l'utilisateur, comme ajouter et supprimer ou taper .
Tous sera défini dans le fichier **script.js**

index.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">

  <title>Tasks</title>

  <link rel="stylesheet" href="style.css">
</head>

<body>
  <div id="root"></div>

  <script src="script.js"></script>
</body>

</html>
```

Exemple2 :MVC

298

Style.css

```

*, [type="text"], [type="text"]:active, [type="text"]:focus {
  ::before, ::after { outline: 0; border: 2px solid #007bff; }
  box-sizing: border-box
}

html {
  font-family: sans-serif;
  font-size: 1rem;
  color: #444;
}

#root {
  max-width: 450px;
  margin: 2rem auto;
  padding: 0 1rem;
}

form {
  display: flex;
  margin-bottom: 2rem;
}

button {
  display: inline-block;
  -webkit-appearance: none;
  padding: .5rem 1rem;
  font-size: 1rem;
  border: 2px solid #ccc;
  border-radius: 4px;
}

button {
  cursor: pointer;
  background: #007bff;
  color: white;
  border: 2px solid #007bff;
  margin: 0 .5rem;
}

[type="text"] {
  width: 100%;
}

li {
  display: flex;
  align-items: center;
  padding: 1rem;
  margin-bottom: 1rem;
  background: #f4f4f4;
  border-radius: 4px;
}

li span {
  display: inline-block;
  padding: .5rem;
  width: 250px;
  border-radius: 4px;
  border: 2px solid transparent;
}

li span:hover {
  background: rgba(179, 215, 255, 0.52);
}

li span:focus {
  outline: 0;
  border: 2px solid #007bff;
  background: rgba(179, 207, 255, 0.52)
}

```

Exemple2 :MVC

299

Script.js

On définit ici notre modèle MVC

```
/**  
 * @class Model  
 *  
 * Gère les données de l'application.  
 */  
class Model {  
    constructor() {  
        this.todos = JSON.parse(localStorage.getItem('todos')) || []  
    }  
  
    bindTodoListChanged(callback) {  
        this.onTodoListChanged = callback  
    }  
  
    _commit(todos) {  
        this.onTodoListChanged(todos)  
        localStorage.setItem('todos', JSON.stringify(todos))  
    }  
}
```

Exemple2 :MVC

300

script.js (la suite)

```
addTodo(todoText) {
  const todo = {
    id: this.todos.length > 0 ? this.todos[this.todos.length - 1].id + 1 : 1,
    text: todoText,
    complete: false,
  }

  this.todos.push(todo)

  this._commit(this.todos)
}

editTodo(id, updatedText) {
  this.todos = this.todos.map(todo =>
    todo.id === id ? { id: todo.id, text: updatedText, complete: todo.complete } : todo
  )

  this._commit(this.todos)
}

deleteTodo(id) {
  this.todos = this.todos.filter(todo => todo.id !== id)

  this._commit(this.todos)
}

toggleTodo(id) {
  this.todos = this.todos.map(todo =>
    todo.id === id ? { id: todo.id, text: todo.text, complete: !todo.complete } : todo
  )

  this._commit(this.todos)
}
```

Exemple :MVC

301

script.js (la suite)

```

/*
 * @class View
 *
 * Représentation visuelle du modèle.
 */
class View {
  constructor() {
    this.app = this.getElement('#root')
    this.form = this.createElement('form')
    this.input = this.createElement('input')
    this.input.type = 'text'
    this.input.placeholder = 'Add todo'
    this.input.name = 'todo'
    this.submitButton = this.createElement('button')
    this.submitButton.textContent = 'Submit'
    this.form.append(this.input, this.submitButton)
    this.title = this.createElement('h1')
    this.title.textContent = 'Todos'
    this.todoList = this.createElement('ul', 'todo-list')
    this.app.append(this.title, this.form, this.todoList)

    this._temporaryTodoText = ''
    this._initLocalListeners()
  }

  get _todoText() {
    return this.input.value
  }

  _resetInput() {
    this.input.value = ''
  }

  createElement(tag, className) {
    const element = document.createElement(tag)

    if (className) element.classList.add(className)

    return element
  }

  getElement(selector) {
    const element = document.querySelector(selector)

    return element
  }

  displayTodos(todos) {
    //Supprimer tous les nœuds
    while (this.todoList.firstChild) {
      this.todoList.removeChild(this.todoList.firstChild)
    }
  }
}

```

Exemple2 :MVC

302

script.js (la suite)

```

    /
    / Afficher le message par défaut
if (todos.length === 0) {
  const p = this.createElement('p')
  p.textContent = 'rien A faire,Ajouter une tache?'
  this.todoList.append(p)
} else {
  // Créer des nœuds
  todos.forEach(todo => {
    const li = this.createElement('li')
    li.id = todo.id

    const checkbox = this.createElement('input')
    checkbox.type = 'checkbox'
    checkbox.checked = todo.complete

    const span = this.createElement('span')
    span.contentEditable = true
    span.classList.add('editable')

    if (todo.complete) {
      const strike = this.createElement('s')
      strike.textContent = todo.text
      span.append(strike)
    } else {
      span.textContent = todo.text
    }
  })
}
  
```

```

const deleteButton = this.createElement('button', 'delete')
deleteButton.textContent = 'Delete'
li.append(checkbox, span, deleteButton)

// Ajouter les nœuds
this.todoList.append(li)
}

// Débogage
console.log(todos)
}

_initLocalListeners() {
  this.todoList.addEventListener('input', event => {
    if (event.target.className === 'editable') {
      this._temporaryTodoText = event.target.innerText
    }
  })
}

bindAddTodo(handler) {
  this.form.addEventListener('submit', event => {
    event.preventDefault()

    if (this._todoText) {
      handler(this._todoText)
      this._resetInput()
    }
  })
}
  
```

Exemple2 :MVC

303

script.js (la suite)

```
}

bindDeleteTodo(handler) {
  this.todoList.addEventListener('click', event => {
    if (event.target.className === 'delete') {
      const id = parseInt(event.target.parentElement.id)

      handler(id)
    }
  })
}

bindEditTodo(handler) {
  this.todoList.addEventListener('focusout', event => {
    if (this._temporaryTodoText) {
      const id = parseInt(event.target.parentElement.id)

      handler(id, this._temporaryTodoText)
      this._temporaryTodoText = ''
    }
  })
}
```

```
bindToggleTodo(handler) {
  this.todoList.addEventListener('change', event => {
    if (event.target.type === 'checkbox') {
      const id = parseInt(event.target.parentElement.id)

      handler(id)
    }
  })
}
```

Exemple2 :MVC

304

script.js (la suite)

```
/*
 * @class Controller
 *
 * Relie l'entrée utilisateur et la sortie de la vue.
 *
 * @param model
 * @param view
 */
class Controller {
  constructor(model, view) {
    this.model = model
    this.view = view

    // Explicite cette liaison
    this.model.bindTodoListChanged(this.onTodoListChanged)
    this.view.bindAddTodo(this.handleAddTodo)
    this.view.bindEditTodo(this.handleEditTodo)
    this.view.bindDeleteTodo(this.handleDeleteTodo)
    this.view.bindToggleTodo(this.handleToggleTodo)

    // Afficher les todos initiaux
    this.onTodoListChanged(this.model.todos)
  }

  onTodoListChanged = todos => {
    this.view.displayTodos(todos)
  }
}
```

```
handleAddTodo = todoText => {
  this.model.addTodo(todoText)
}

handleEditTodo = (id, todoText) => {
  this.model.editTodo(id, todoText)
}

handleDeleteTodo = id => {
  this.model.deleteTodo(id)
}

handleToggleTodo = id => {
  this.model.toggleTodo(id)
}

const app = new Controller(new Model(), new View())
```

Créer le formulaire suivant qui permet de stocker les données sur localStorage et 'afficher sur une liste.

1. Créer classe User avec constructeur name et email
2. Créer classe UserForm de constructeur form et liste qui contient une méthodes qui permet de stocker les données **saveUsers** ,une méthode submit () qui envoie les données ,une méthode addUser(user) et deleteUser(user)
3. Créer l'instance de userForm et appliquer sur le form.

Name: Email: Save

- aaaa (bbbbbb@gmail.com)
- hassan@ihass.com (hassan@ihass.com)

Créer une application de calculatrice qui utilise l'architecture MVC. La vue devrait avoir des boutons pour les opérations de base (addition, soustraction, multiplication, division) et des champs pour entrer les nombres. Le modèle devrait gérer les calculs, et le contrôleur devrait gérer les interactions utilisateur.

Calculatrice

Premier nombre :

Deuxième nombre :

+ - * /

Résultat :

Créer une application de liste de tâches qui utilise l'architecture MVC. La vue devrait afficher une liste de tâches à faire et un champ pour ajouter de nouvelles tâches. Le modèle devrait contenir les tâches à faire, et le contrôleur devrait gérer l'ajout et la suppression de tâches. Les tâches devraient être stockées dans le local storage

Liste de tâches

Nouvelle tâche : Ajouter

Partie 11 :

JavaScript Asynchrone

-fonction callback

-promise

-promise.all

-Async/Await

-Fetch

-formData

-Ajax

Fonction de Rappel: Callback

307

- Une fonction de rappel (callback en anglais) est une fonction passée en paramètre d'une autre fonction en tant qu'argument ;
- • La fonction de rappel est invoquée à l'intérieur de la fonction externe pour exécuter des instructions précises.

```
function affichage(s) {
  console.log(s);
}

function calcul(num1, num2, myCallback) {
  let somme = num1 + num2;
  myCallback(somme);
}

calcul(1, 5, affichage);
```

Dans l'exemple, « affichage » est le nom d'une fonction qui est passée à la fonction calcul() comme argument.
ne pas utiliser les parenthèses dans l'appel de la fonction.

```
function add(a,b){
  return a+b; }

function divide(a,b){
  return a/b;
}

function calcul(x,y,callback){ //callback
  return callback(x,y);
}

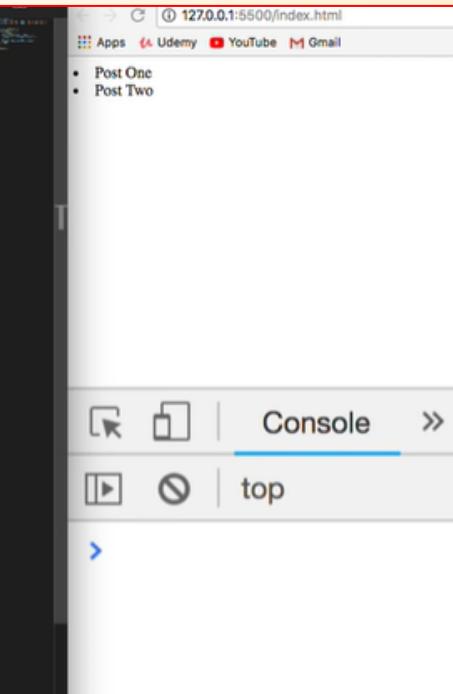
console.log(calcul(2,3,add)); 5
console.log(calcul(2,3,divide)); 0.666666
```

Exemple fonction de Rappel

```

1 const posts = [
2   { title: 'Post One', body: 'This is post one' },
3   { title: 'Post Two', body: 'This is post two' }
4 ];
5
6 function getPosts() {
7   setTimeout(() => {
8     let output = '';
9     posts.forEach((post, index) => {
10       output += `<li>${post.title}</li>`;
11     });
12     document.body.innerHTML = output;
13   }, 1000);
14 }
15
16 getPosts();
17

```

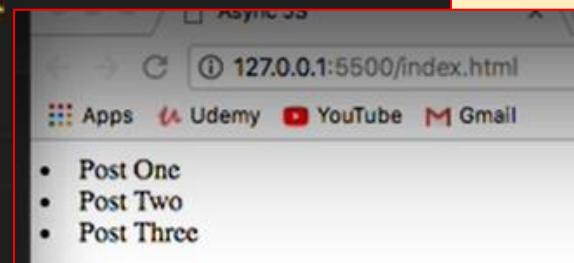


On veut créer une méthode qui cree un post et affiche les posts avant l'execusion de la premiere méthode getPost ,on utilise donc Fonction Callback

```

5 function createPost(post, callback) {
6   setTimeout(() => {
7     posts.push(post);
8     callback();★
9   }, 2000);
10 }
11
12 createPost({ title: 'Post Three', body: 'This is post three' }, getPosts);★

```



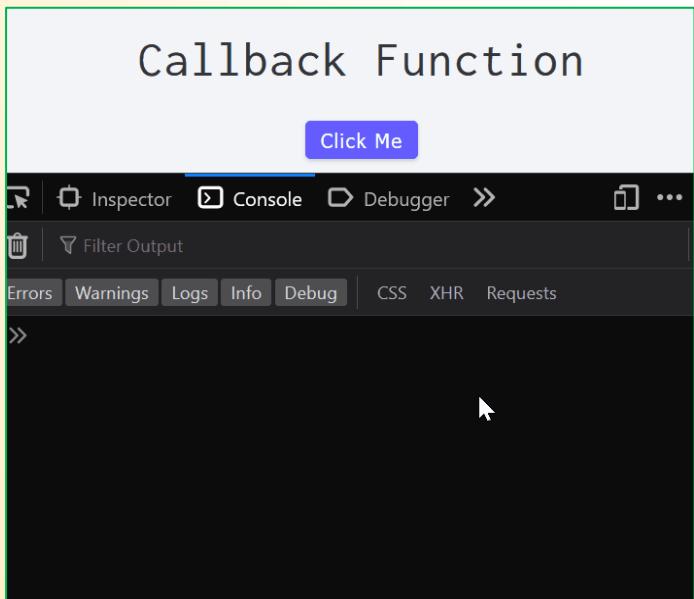
EP44:

309

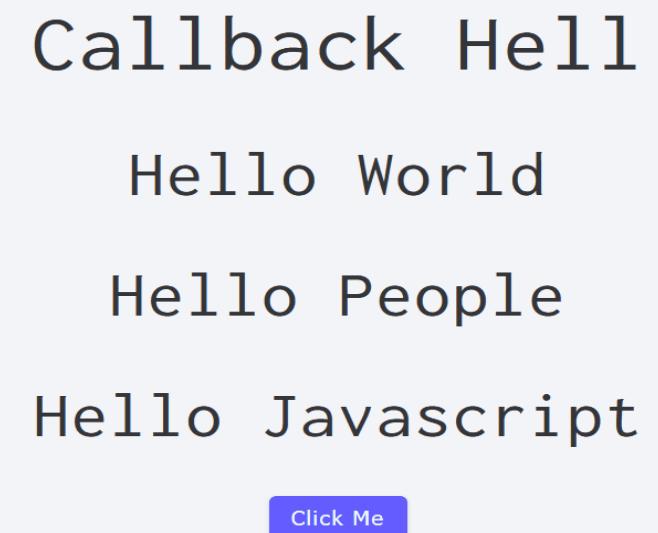
Créez une fonction appelée **doubleArray(arr, callback)** qui prend un tableau de nombres comme argument et renvoie un nouveau tableau dans lequel chaque nombre est doublé. Utilisez la méthode `forEach()` et une fonction de rappel pour implémenter la logique.

Créez une fonction appelée **getEvenNumbers(arr, callback)** qui prend un tableau de nombres comme argument et renvoie un nouveau tableau contenant uniquement les nombres pairs. Utilisez la méthode `filter()` et une fonction de rappel pour implémenter la logique.

Créer deux évènements sur un button pour afficher deux messages Hello world et hello people ,le 2eme c'est un appel fonction callback



Créer un évènement qui appelle `settimeout` imbriqué pour colorer 3 divs au bout des intervalles .



Promise

310

Une *promesse* (promise) est un objet spécial en javascript qui lie le “producteur de code” et le “consommateur de code” ensemble. En comparant à notre analogie c'est la “liste d'abonnement”. Le “producteur de code” prends le temps nécessaires pour produire le résultat promis, et la “promesse” donne le résultat disponible pour le code abonné quand c'est prêt.

En JavaScript, une promesse a trois états :

- En attente (Pending) : résultat indéfini ;
- Accompli (Fulfilled) : opération réussie, le résultat est une valeur ;
- Rejeté (Rejected) : le résultat est une erreur.

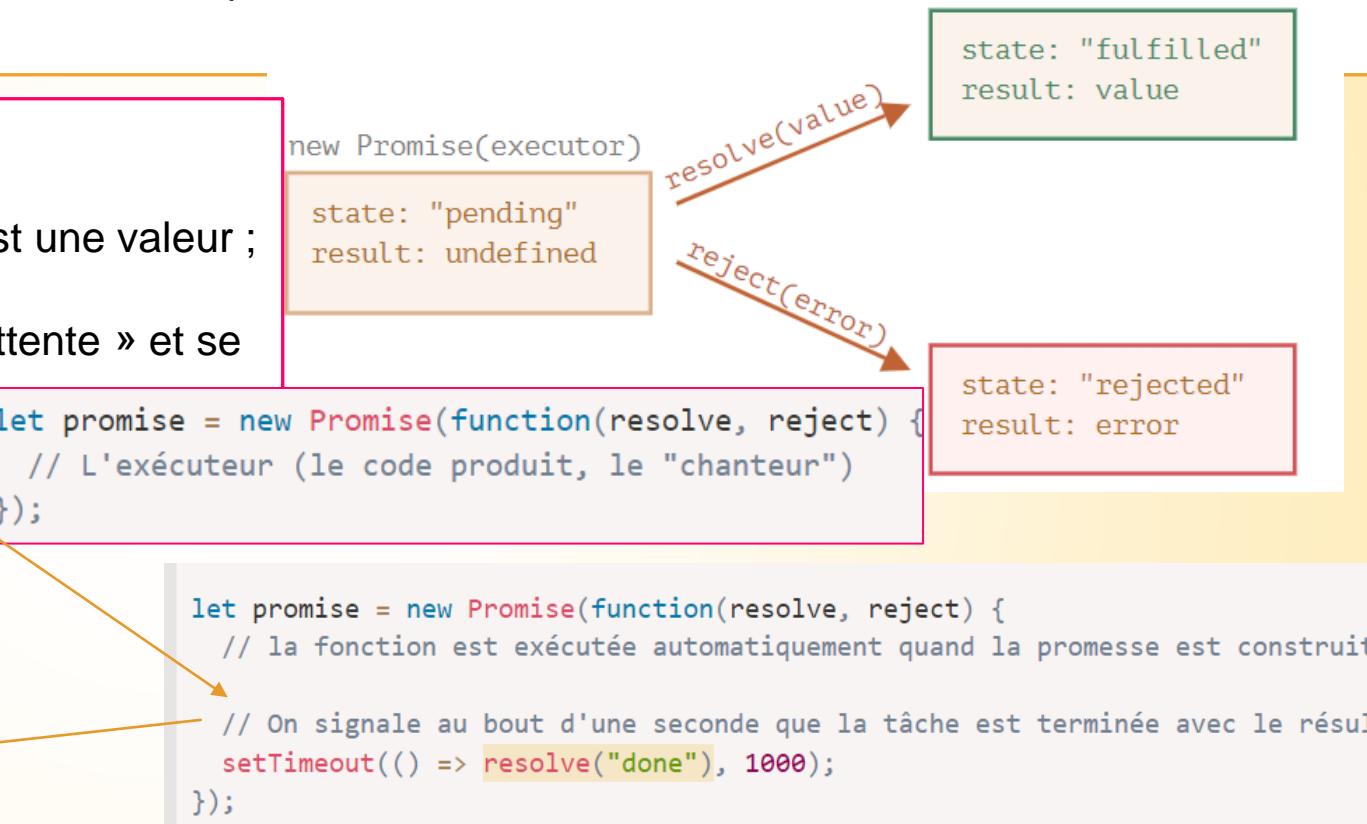
Une promesse commence toujours dans l'état « en attente » et se termine par un des états « accompli » ou « rejeté »

Créer une promesse

```
new Promise((resolve, reject) => {
  setTimeout(() => resolve("result"), 2000)
})
  .finally(() => alert("Promise ready"))
  .then(result => alert(result)); // <-- .then gère le résultat
```

avec une erreur dans la promesse, passant à travers `finally` vers `catch` :

```
new Promise((resolve, reject) => {
  throw new Error("error");
})
  .finally(() => alert("Promise ready"))
  .catch(err => alert(err)); // <-- .catch gère l'objet error
```



```
let promise = new Promise(function(resolve, reject) {  
  // On signale après 1 seconde que la tâche est terminée avec une erreur  
  setTimeout(() => reject(new Error("Whoops!")), 1000);  
});
```

Promise :resolve, reject, then ,catch

311

```
const p=new Promise((resolve,reject)=>{
    resolve(new Date().toLocaleTimeString());
});
console.log(p);  Promise { '14:25:42' }
//si la promesse est résolu :
p.then((n)=>{
    console.log("la date est",n )  la date est 14:25:42
});
//si l'échec :
p.catch((e)=>{
    console.log("RDV rejeté",e);
});
```

```
//!Une promesse qui se resolve en devinant un nombre.
const value = 3
const promise = new Promise((resolve, reject) => {
    const random = Math.floor(Math.random() * 3)
    console.log(random)  2
    if (random === value) {
        resolve('Vous avez deviné correctement')
    } else {
        reject('mauvais numéro')
    }
})
console.log(promise)  Promise { <rejected> 'mauvais numéro' }
promise.then((data) => console.log(data))
    .catch((err) => console.log(err))
```

Enchaîner les promises

```
//!enchaîner les promesses
const p2=new Promise((resolve,reject)=>{
    resolve("200DH");
});
p2.then((n)=>{
    console.log(n); 200DH
    return "100DH" //2eme enchainement
}).then((n)=>console.log(n))
    .catch((e)=>{
        console.log("erreur",e);
    })
    .finally(()=>console.log("done"));
```

Promise:Exemple1

mohamed@goumih.com

313

- Au lieu d'utiliser une méthode callback on utilise une promesse :dans l'exemple de fonction de rappel

```
function createPost(post){  
    return new Promise((resolve,reject)=>{  
        setTimeout(()=>{  
            posts.push(post);  
            const error=false;  
            if(!error){  
                resolve();  
                getPosts();  
            }else{  
                reject("error")  
            }  
        },2000)  
    })  
}  
  
createPost({title:'Post Three',body:'this is post three'})  
.then((data)=>console.log("done"))  
.catch((error)=>console.log(error))
```

Promise :Exemple2

314

```
function attendre(duration){  
    return new Promise((resolve)=>{  
        setTimeout(()=>{  
            resolve(duration)  
        },duration)  
    })  
}  
  
function attendreAndStop(duration){  
    return new Promise((reject)=>{  
        setTimeout(()=>{  
            reject(duration)  
        },duration)  
    })  
}
```

```
attendre(2000)  
.then(()=>{  
    console.log('attente 2s')  attente 2s  
    return attendre(1000);  
})  
.then(()=>{  
    console.log('attente 1 s')  attente 1 s  
    attendreAndStop(1000)  
})  
.catch((e)=>{  
    console.log("erreur",e)  
})  
.finally(()=>console.log("stop"))
```

Promise.all

315

Promise.all permet d'exécuter plusieurs promises à la fois

```
const promise1=Promise.resolve("hello world");
const promise2=10;
const promise3=new Promise((resolve,reject)=>
setTimeout(resolve,2000,"Goodbye")
);
//promise.all
Promise.all([promise1,promise2,promise3])
.then(values=>console.log(values));
```

Async/await

316

Le mot “**async**” devant une fonction signifie une chose simple : une fonction renvoie toujours une promesse.

Par exemple, cette fonction renvoie une promesse résolue avec le résultat 1 ;

Nous pourrions explicitement renvoyer une promesse, ce qui reviendrait au même :

```
async function f() {
    return 1;
}

f().then(alert); // 1
```

Le mot-clé **await** fait en sorte que JavaScript attende que cette promesse se réalise et renvoie son résultat. Voici un exemple avec une promesse qui se résout en 1 seconde :

```
async function f() {
    return Promise.resolve(1);
}

f().then(alert); // 1
```

```
async function f() {
    let promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve("done!"), 1000)
    });
    let result = await promise; // attendre que la
    // promesse soit résolue (*)
    alert(result); // "done!"
}
f();
```

createPost() avec function async/await

- On reprend notre exemple de createPost
- On a utiliser précédemment callback et promesse
- Cette fois on va utiliser créer une fonction init async init() qui appelle les deux fonctions createPost et getPosts()

```
// Async / Await
async function init() {
    await createPost({ title: 'Post Three', body: 'This is
post three' });

    getPosts();
}
```

- Post One
- Post Two
- Post Three

Fetch:

318

La fonction **fetch** permet de Charger des informations à partir d'un serveur :

- // API du navigateur pour les demandes HTTP (AJAX) : sans recharger la page
- // Obtenez les demandes, ... // retourne promise
- Syntaxe:

```
let response = await fetch(url);

if (response.ok) { // if HTTP-status is 200-299
  // obtenir le corps de réponse (la méthode expliquée ci-dessous)
  let json = await response.json();
} else {
  alert("HTTP-Error: " + response.status);
}
```

Response fournit plusieurs méthodes basées sur les promesses pour accéder au corps dans différents formats :

- response.text()** – lit la réponse et retourne sous forme de texte,
- response.json()** – analyse la réponse en JSON,

```
//on utilise await avec async
async function getUsers(){
  result=fetch('https://jsonplaceholder.typicode.com/users')
  let response=await result;
  let data=response.json();
  console.log(data)
}
getUsers()
```

Fetch: lire les fichiers

319

la même exemple précédent sans **await**, en utilisant la syntaxe des promesses pures :

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response=>response.json())
  .then(data=>console.log(data[0].address.city))
```

Pour obtenir la réponse en texte, en utilisent **response.text()** : ici on lit un fichier text

```
fetch("test.txt")
  .then(response=>response.text())
  .then(data=>console.log(data))
```

≡ test.txt X
codes_cours > Async_promis
1 dev101 dev102

Le format JSON est utilisé la plupart du temps. Par exemple: on lit un fichier json



```
urs > Fetch > {} user.json > ...
{
  "users": [
    {"name": "John", "surname": "Smith"},  

    {"name": "John", "surname": "Dow"},  

    {"name": "mohamed", "surname": "goumih"}
  ]
}
```

```
fetch("user.json")
  .then(response=>response.json())
  .then(data=>showUser(data));
function showUser(data){
  console.table(data.users);
}
```

Envoi du formulaire avec formData

formData permet d'envoyer le corps d'un formulaire :

```
new FormData(formElem)
```

```
<form id="formElem">
  <input type="text" name="name" value="mohamed">
  <input type="text" name="surname" value="Ahmed">
  <input value="envoyer" type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('https://jsonplaceholder.typicode.com/users', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    console.log(result);
  };
</script>
```

Fetch: méthode POST

321

Pour faire une requête POST, ou une requête avec une autre méthode, nous devons utiliser les options fetch :

- **method** – HTTP-method, par exemple POST,
- **body** – le corps de la requête, un parmi ceux-ci :
 - une chaîne de caractères (par exemple encodé en JSON),
 - un objet FormData, pour soumettre les données par Form,

```
let user = {  
    name: 'John',  
    surname: 'Smith'  
};  
fetch('https://jsonplaceholder.typicode.com/users', {  
    method: 'POST',  
    headers: {  
        'Content-Type': 'application/json; charset=utf-8'  
    },  
    body: JSON.stringify(user)  
})  
.then(response=>response.json())  
.then(data=>console.log(data))
```

Si la requête body est une chaîne de caractères, alors l'en-tête Content-Type est défini sur **text/plain; charset=UTF-8** par défaut.

Si JSON, nous utiliserons à la place l'option headers pour envoyer application/json, le bon Content-Type pour les données encodées en JSON.

Exemple fetch,fromData,php

322

```
<form id="myForm" method="post" >
  <label for="name">Nom :</label>
  <input type="text" id="name" name="name">

  <label for="email">Email :</label>
  <input type="email" id="email" name="email">

  <button type="submit">Envoyer</button>
</form>
<script>
  const myForm = document.getElementById('myForm');
myForm.addEventListener('submit', event => {
  //event.preventDefault();
  const formData = new FormData(myForm);
  fetch('fetch_form.php', {
    method: 'POST',
    body: formData
  })
  .then(response => response.text())
  .then(data => console.log(data))
  .catch(error => console.error(error));
});
</script>
```

```
<?php
if (isset($_POST['name']) && isset($_POST['email'])) {
  $name = $_POST['name'];
  $email = $_POST['email'];

  echo "Le nom est : $name\n";
  echo "L'email est : $email\n";
} else {
  echo "Erreur : les champs 'name' et 'email' sont obligatoires.";
}
?>
```

Exemple :fetch,php,mysql

323

Fetch_bd.php

```
<?php
// Connexion à La base de données
$db_host = 'localhost';
$db_name = 'fetch';
$db_user = 'root';
$db_pass = '';
$db = new PDO("mysql:host=$db_host;dbname=$db_name;charset=utf8", $db_user, $db_pass);

// Récupération des données de La base de données
$query = "SELECT id, name,email FROM users";
$stmt = $db->query($query);
$users = $stmt->fetchAll(PDO::FETCH_ASSOC);

// Génération du HTML pour le select
$select_html = '<select name="user">';
foreach ($users as $user) {
    $select_html .= '<option value="' . $user['id'] . '">' . $user['name'] . '</option>';
}
$select_html .= '</select>';

// Envoi du HTML en réponse à La requête Fetch
echo $select_html;
```

On affiche la liste des noms de Base donnée dans une page php et on les récupèrent avec fetch

```
<div id="user-select"></div>
<script>
    fetch('fetch_bd.php')
    .then(response => response.text())
    .then(html => {
        document.getElementById('user-select').innerHTML = html;
    })
    .catch(error => console.error(error));

</script>
```

EP45:

324

Créez une fonction qui renvoie une Promise qui se resolve après un certain nombre de secondes.
Utiliser second comme argument et setTimeout

Écrivez une fonction qui renvoie une promesse qui se résout en un nombre aléatoire compris entre 1 et 100.Si le nombre est pair, sinon reject.

Écrivez une fonction qui renvoie une promesse qui se résout en résultat de l'appel de fetch de lien
<https://example.com/api>

Créez une fonction qui renvoie une promesse qui se résout en un tableau de nombres aléatoires entre 1 et 10.
En stocke les valeurs dans le tableau et on retourne le résultat avec Promise.all.

```
const users = [
  { id: 1, name: 'john' },
  { id: 2, name: 'susan' },
  { id: 3, name: 'anna' },
]

const articles = [
  { userId: 1, articles: [ 'one', 'two', 'three' ] },
  { userId: 2, articles: [ 'four', 'five' ] },
  { userId: 3, articles: [ 'six', 'seven', 'eight', 'nine' ] },
]
```

- 1.Créer une fonction de promesse **getUser(name)** qui retourne un user par son nom .
- 2.Créer une fonction de promesse **getArticles(userId)** qui retourne la liste des articles d'un user par son userID.
- 3.créer une fonction **async getData(name)** qui permet de trouver tous les articles d'un user,la fonction doit appeler les autres fonctions

EP46:

325

Réécrire cet exemple de code en utilisant `async/await` au lieu de `.then/catch`:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    });
}

loadJson('https://javascript.info/no-such-user.json')
  .catch(alert); // Error: 404
```

Créez une fonction asynchrone `getUsers(names)`, qui obtient un tableau de connexions de **GitHub** ou **jsonplaceholder.typicode.com**, récupère les utilisateurs de GitHub et renvoie un tableau d'utilisateurs GitHub. L'URL GitHub avec les informations utilisateur pour la donnée `USERNAME` est : <https://api.github.com/users/USERNAME> Ou <https://jsonplaceholder.typicode.com/users/2>

TP11:

326

Utiliser fetch pour afficher la liste de 5 utilisateurs de <https://jsonplaceholder.typicode.com/users> dans un tableau HTML :

Nom	Email	Site Web
Leanne Graham	Sincere@april.biz	hildegard.org
Ervin Howell	Shanna@melissa.tv	anastasia.net
Clementine Bauch	Nathan@yesenia.net	ramiro.info
Patricia Lebsack	Julianne.OConner@kory.org	kale.biz
Chelsey Dietrich	Lucio_Hettinger@annie.ca	demarco.info

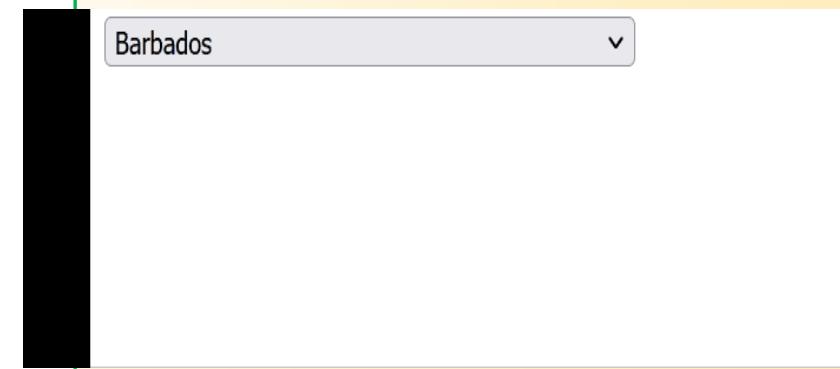
Utiliser promises avec fetch pour afficher la liste de 5 commentaires de <https://jsonplaceholder.typicode.com/comments> dans une liste ul:
Créer une fonction addComment pour ajouter un commentaire.

Liste des commentaires

- id labore ex et quam laborum (Eliseo@gardner.biz): laudantium enim quasi est quidem magnam voluptate ipsam eos tempora qui sapiente accusantium
- quo vero reiciendis velit similique earum (Jayne_Kuhic@sydney.com): est natus enim nihil est dolore omnis voluptatem numquam pariatur nihil sint nostrum voluptatem reiciendis et
- odio adipisci rerum aut animi (Nikita@garfield.biz): quia molestiae reprehenderit quasi aspernatur aut expedita occaecati aliquam accusamus maiores nam est cum et ducimus et vero voluptates excepturi deleniti ratione
- alias odio sit (Lew@alysha.tv): non et atque occaecati deserunt quas accusantium unde odit nobis qui voluptatem quia voluptas et
- vero eaque aliquid doloribus et culpa (Hayden@althea.biz): harum non quasi et ratione tempore iure ex voluptates in ratione harum et

Nom : Email : Message : Ajouter un commentaire

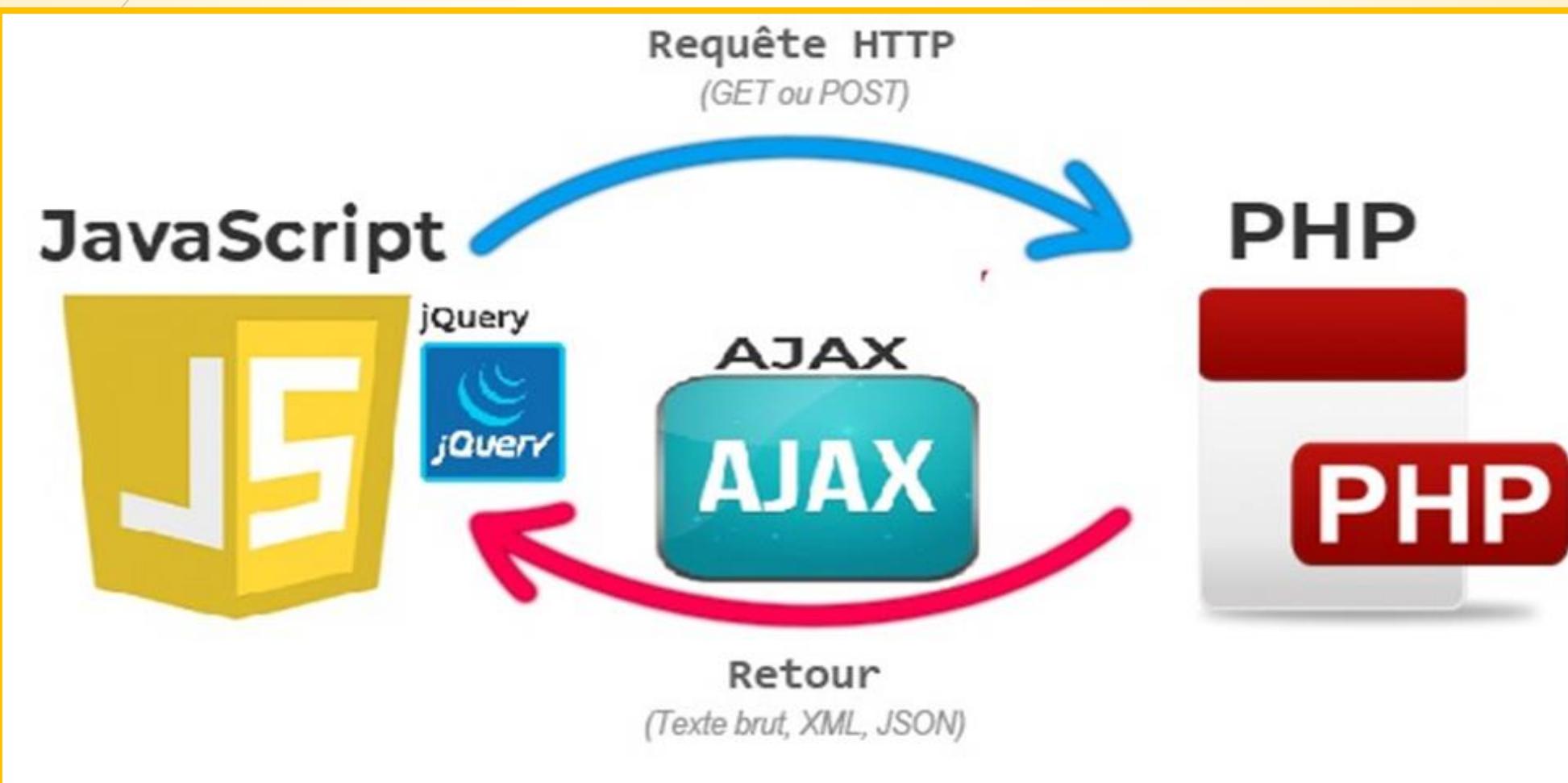
- Créer une fonction `generateOption(text,value)` qui permet de retourner une option avec text et value .
- Créer une fonction `showInfo(countries)` qui permet créer un select dans le body contenant un country.name.common et le value country.flags.png ,la fonction doit appeler `generateOption` .
- Ajouter une image pour afficher les flags
- Utiliser fetch pour récupérer la liste des pays de <https://restcountries.com/v3.1/all> et les afficher avec la fonction `showInfo()`.



AJAX

327

On essaye de créer des applications en communication avec le serveur



Ajax

328

- ▶ AJAX est l'art d'échanger des données avec un serveur et de mettre à jour des parties d'une page Web - sans recharger toute la page.
- ▶ **AJAX = JavaScript et XML asynchrones.**
- ▶ Bien qu'il ait le mot “XML” dans son nom, il peut fonctionner sur toutes les données, pas seulement au format XML
- ▶ En bref; AJAX consiste à charger des données en arrière-plan et à les afficher sur la page Web, sans recharger toute la page.
- ▶ Exemples d'applications utilisant AJAX : onglets Gmail, Google Maps, Youtube et Facebook.
- ▶ AJAX n'est pas un langage de programmation.
- ▶ AJAX utilise simplement une combinaison de :
 - Un objet XMLHttpRequest intégré au navigateur (pour demander des données à un serveur Web)
 - JavaScript et HTML DOM (pour afficher ou utiliser les données).
 - **À l'heure actuelle, il existe la méthode moderne, fetch, qui déprécie quelque peu XMLHttpRequest.**

Les étapes pour utiliser AJAX(ES6)

mohamed@goumih.com

329

► **XMLHttpRequest** est un objet intégré du navigateur qui permet de faire des requêtes HTTP en JavaScript.

► **XMLHttpRequest** a deux modes de fonctionnement : synchrone et asynchrone. Voyons l'asynchrone, car il est utilisé dans la majorité des cas.
Pour faire la requête, nous avons besoin de 4 étapes :

1.Créer XMLHttpRequest:

```
let xhr = new XMLHttpRequest();
```

2.L'initialiser:

```
xhr.open(method, URL, [async, user, password])
```

3.l'envoyer:

```
xhr.send([body])
```

Cette méthode ouvre la connexion et envoie la demande au serveur. Le paramètre facultatif body contient le corps de la requête.

Certaines méthodes de requête comme GET n'ont pas de corps. Et certains d'entre eux comme POST utilisent body pour envoyer les données au serveur.

4.Écouter les événements xhr pour obtenir une réponse.

3événements sont les plus utilisés :

- **load** – lorsque la requête est terminée (même si l'état HTTP est de type 400 ou 500) et que la réponse est entièrement téléchargée.
- **error** – lorsque la requête n'a pas pu être faite, par exemple réseau en panne ou URL non valide.
- **progress** – se déclenche périodiquement pendant le téléchargement de la réponse, indique combien a été téléchargé.

```
xhr.onload = function() {  
    alert(`Loaded: ${xhr.status} ${xhr.response}`);  
};  
  
xhr.onerror = function() {  
    alert(`Network Error`);  
};  
  
xhr.onprogress = function(event) {
```

• **method** – Méthode HTTP.

Habituellement "GET" ou "POST".

• **URL** – l'URL à demander, une chaîne de caractères,

• **async** – si explicitement défini sur false, alors la demande est synchrone

• **user, password** – identifiant et mot de passe pour l'authentification HTTP de base (si nécessaire).

Veuillez noter que l'appel **open**, contrairement à son nom, n'ouvre pas la connexion. Il configure uniquement la demande, mais l'activité réseau ne démarre qu'avec l'appel de **send**.

Réponses XHR (ES6)

330

Une fois que le serveur a répondu, nous pouvons recevoir le résultat dans les propriétés **xhr** suivantes :
status

Code d'état HTTP (un nombre): 200, 404, 403 et ainsi de suite, peut être 0 en cas d'échec non-HTTP.

statusText :

Message d'état HTTP (une chaîne de caractères): généralement OK pour 200, Not Found pour 404, Forbidden pour 403 et ainsi de suite.

response (les anciens scripts peuvent utiliser responseText)

Le corps de réponse du serveur.

Nous pouvons également spécifier un délai d'expiration en utilisant la propriété correspondante :

```
xhr.timeout = 1000; // délai d'attente en ms, 10 secondes
```

Nous pouvons utiliser la propriété **xhr.responseText** pour définir le format de réponse :

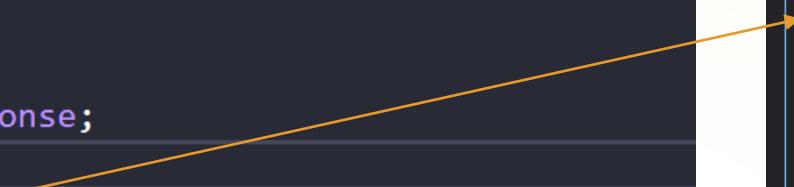
- "" (default) – obtenir en tant que chaîne de caractères,
- "text" – obtenir en tant que chaîne de caractères,
- "document" – obtenir en tant que document XML) ou document HTML
- "json" – obtenir en tant que JSON (analysé automatiquement).
-

Exemple de Réponse Json et text (ES6)

331

```

let xhr = new XMLHttpRequest();
xhr.open('GET', 'https://jsonplaceholder.typicode.com/users');
xhr.responseType = 'json';
xhr.send();
xhr.onload = function() {
  let responseObj = xhr.response;
  console.log(responseObj);
};
  
```

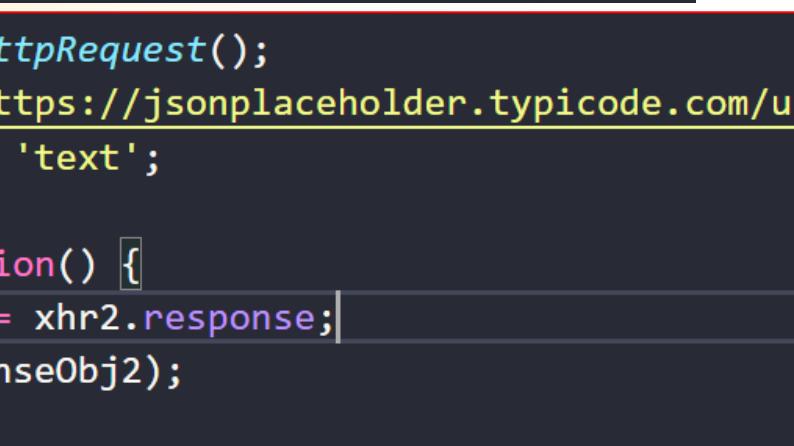


```

[{"id": 1, "name": "Leanne Graham", "username": "Bret", "email": "Sincere@april.biz", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 2, "name": "Ervin Howell", "username": "Antonette", "email": "Shanna@melissa.tv", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 3, "name": "Clementine Bauch", "username": "Samantha", "email": "Natalie@kelly.ca", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 4, "name": "Patricia Lebsack", "username": "Karianne", "email": "Doris@armstrong.net", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 5, "name": "Chelsey Dietrich", "username": "Kamren", "email": "Moriah.Stanton@melissa.tv", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 6, "name": "Mrs. Dennis Schulist", "username": "Leopoldo_Corkery", "email": "Karley_Davis@april.biz", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 7, "name": "Kurtis Weissnat", "username": "Elwyn.Skiles", "email": "Tatyana_Kramer@april.biz", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 8, "name": "Nicholas Runolfsdottir V", "username": "Maxime_Nienow", "email": "Glenna Reichert@april.biz", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 9, "name": "Glenna Reichert", "username": "Delphine", "email": "Tatyana_Kramer@april.biz", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, {"id": 10, "name": "Clementina DuBuque", "username": "Moriah.Stanton", "email": "Karley_Davis@april.biz", "address": {"street": "Kulas Light", "suite": "Apt. 556", "city": "Gwenborough", "zipcode": "92998-3874", "geo": {"lat": "-37.3159", "lng": "81.1496"}}, "length": 10]
  
```

```

let xhr2 = new XMLHttpRequest();
xhr2.open('GET', 'https://jsonplaceholder.typicode.com/users');
xhr2.responseType = 'text';
xhr2.send();
xhr2.onload = function() {
  let responseObj2 = xhr2.response;
  console.log(responseObj2);
};
  
```



```

{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  ...
}
  
```

Dans les anciens scripts, vous pouvez également trouver des propriétés `xhr.responseText` et même `xhr.responseXML`. Ils existent pour des raisons historiques, pour obtenir une chaîne de caractères ou un document XML. De nos jours, nous devons définir le format dans `xhr.responseType` et obtenir `xhr.response` comme illustré ci-dessus.

Etats prêts (ES6)

332

XMLHttpRequest change entre les états au fur et à mesure de sa progression.

L'état actuel est accessible en tant que **xhr.readyState**.

```
UNSENT = 0; // état initial
OPENED = 1; // open appelé
HEADERS_RECEIVED = 2; // en-têtes de réponse reçus
LOADING = 3; // la réponse est en cours de chargement
DONE = 4; // requête terminée
```

Un objet **XMLHttpRequest** voyagent dans l'ordre **0 → 1 → 2 → 3 → ... → 3 → 4**. L'état **3** se répète chaque fois qu'un paquet de données est reçu sur le réseau.

Nous pouvons les suivre en utilisant l'événement **readystatechange** :

Vous pouvez trouver des écouteurs **readystatechange** dans un code très ancien, il est là pour des raisons historiques, car il fut un temps où il n'y avait pas de **load** et d'autres événements. De nos jours, les gestionnaires **load/error/progress** le déprécient.

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 3) {
    // chargement
  }
  if (xhr.readyState == 4) {
    // requête terminée
  }
};
```

Nous pouvons mettre fin à la requête à tout moment. L'appel à **xhr.abort()** fait cela : Cela déclenche l'événement **abort** et **xhr.status** devient **0**

```
xhr.abort(); // met fin à la requête
```

Ajax: envoi des requêtes

333

Pour envoyer une requête à un serveur, nous utilisons les méthodes open() et send() de l'objet XMLHttpRequest :

```
xhttp.open("GET", "ajax_info.txt", true);  
xhttp.send();
```

GET est plus simple et plus rapide que POST et peut être utilisé dans la plupart des cas.

Cependant, utilisez toujours les requêtes POST lorsque :

- Un fichier en cache n'est pas une option (mettre à jour un fichier ou une base de données sur le serveur).
 - Envoi d'une grande quantité de données au serveur (POST n'a pas de limitation de taille).
 - En envoyant des entrées utilisateur (qui peuvent contenir des caractères inconnus), POST est plus robuste et sécurisé que GET.
- Dans l'exemple ci-dessus, vous pouvez obtenir un résultat mis en cache. Pour éviter cela, ajoutez un identifiant unique à l'URL :

```
xhttp.open("GET", "demo_get.php?t=" + Math.random(), true);  
xhttp.send();
```

Si vous souhaitez envoyer des informations avec la méthode GET, ajoutez les informations à l'URL :

```
xhttp.open("GET", "demo_get2.php?fname=Henry&lname=Ford", true);  
xhttp.send();
```

requête POST :

Pour POSTer des données comme un formulaire HTML, ajoutez un en-tête HTTP avec setRequestHeader(). Spécifiez les données que vous souhaitez envoyer dans la méthode send() :

```
xhttp.open("POST", "demo_post2.php", true);  
xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");  
xhttp.send("fname=Henry&lname=Ford");
```

Méthode open(méthode ,url ,aysc)

334

```
xhttp.open("GET", "ajax_test.php", true);
```

GET ou POST

Le paramètre url de la méthode open() est une adresse vers un fichier sur un serveur :

Les requêtes du serveur doivent être envoyées de manière asynchrone.
Le paramètre async de la méthode open() doit être défini sur **true** :

En envoyant de manière asynchrone, le JavaScript n'a pas à attendre la réponse du serveur, mais peut à la place :

- exécuter d'autres scripts en attendant la réponse du serveur
- traiter la réponse une fois que la réponse est prête

La propriété onreadystatechange

Avec l'objet XMLHttpRequest, vous pouvez définir une fonction à exécuter lorsque la requête reçoit une réponse.

La fonction est définie dans la propriété **onreadystatechange** de l'objet XMLHttpRequest

```
xhttp.onreadystatechange = function() {
  if (this.readyState == 4 && this.status ==
200) {
    document.getElementById("demo").innerHTML =
      this.responseText;
  }
}
```

AJAX - Réponse du serveur

335

La propriété **onreadystatechange** définit une fonction à exécuter lorsque le readyState change.

La propriété **status** et la propriété **statusText** contiennent le statut de l'objet XMLHttpRequest.

onreadystatechange Defines a function to be called when the readyState property changes

readyState Holds the status of the XMLHttpRequest.

- 0: request not initialized
- 1: server connection established
- 2: request received
- 3: processing request
- 4: request finished and response is ready

requete non initialisée
connexion au serveur créée
demande reçue
traitement de la demande
la demande est terminée et la réponse est prête

status 200: "OK"

OK : l'envoi est fait

403: "Forbidden"

Interdit

404: "Page not found"

Page non trouvé

responseText obtenir les données de réponse sous forme de chaîne

```
document.getElementById("demo").innerHTML = xhttp.responseText;
```

Ajax –réponse du serveur

336

responseXML récupérer les données de réponse sous forme de données XML

```

<script>
var xhttp, xmlDoc, txt, x, i;
xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
if (this.readyState == 4 && this.status == 200) {
xmlDoc = this.responseXML;
txt = "";
x = xmlDoc.getElementsByTagName("name");
for (i = 0; i < x.length; i++) {
txt = txt + x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("demo").innerHTML = txt;
}
};
xhttp.open("GET", "personne.xml", true);
xhttp.send();
</script>
```

La méthode **getAllResponseHeaders()** renvoie toutes les informations d'en-tête de la réponse du serveur.

comme :length, server-type, content-type, last-modified, etc:

```

if (this.readyState == 4 && this.status == 200
) {document.getElementById("demo").innerHTML =
this.getAllResponseHeaders();}
```

La méthode **getResponseHeader()** renvoie des informations d'en-tête spécifiques à partir de la réponse du serveur.

```

if (this.readyState == 4 && this.status == 200
) {
document.getElementById("demo").innerHTML =
this.getResponseHeader("Last-Modified");}
```

Ajax avec PHP

337

L'exemple suivant montre comment une page Web peut communiquer avec un serveur Web pendant qu'un utilisateur saisit des caractères dans un champ de saisie :

```
index.php
+
5 <form action="">
6 Nom: <input type="text" id="txt1" onkeyup="showHint(this.value)">
7 </form>
8 <p>Suggestions: <span id="txtHint"></span></p>
9 <script>
0 function showHint(str) {
1     var xhttp;
2     if (str.length == 0) {
3         document.getElementById("txtHint").innerHTML = "";
4         return;
5     }
6     xhttp = new XMLHttpRequest();
7     xhttp.onreadystatechange = function() {
8         if (this.readyState == 4 && this.status == 200) {
9             document.getElementById("txtHint").innerHTML = this.responseText;
0         }
1     };
2     xhttp.open("GET", "ajax.php?q="+str, true);xhttp.send();    }
3 </script>
```

```
ajax.php
1 <?php
2 $a[] = "Mohamed";$a[] = "Mounir";$a[] = "Wail";$a[] = "Hicham";$a[] = "Aziz";
3 $a[] = "Fatima";$a[] = "Sara";$a[] = "Hamaza";$a[] = "Hind";
4
5 $q = $_REQUEST["q"];
6 $hint = "";
7 if ($q !== "") {
8     $q = strtolower($q);
9     $len=strlen($q);
10    foreach($a as $name) {
11        if (stristr($q, substr($name, 0, $len))) {
12            if ($hint === "") {
13                $hint = $name;
14            } else {
15                $hint .= ", $name";
16            }
17        }
18    }
19 }
20 echo $hint === "" ? "no suggestion" : $hint;
```

Exemple de base de données avec AJAX

index2.php X

```
<form action="">
<select name="customers" onchange="showEtudiant(this.value)">
<option value="">Select a etudiant:</option>
<option value="1">Mohamed </option>
<option value="2">Hicham</option>
<option value="3">Sara</option>

</select>
</form>
<br>
<div id="txtHint">information de l'etudiant...</div>

<script>
function showEtudiant(str) {
    var xhttp;
    if (str == "") {
        document.getElementById("txtHint").innerHTML = "";
        return;
    }
    xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("txtHint").innerHTML = this.responseText;
        }
    };
    xhttp.open("GET", "ajax2.php?q="+str, true);
    xhttp.send();
}
</script>
```

L'exemple suivant montre comment une page Web peut récupérer des informations à partir d'une base de données avec AJAX :

ajax2.php X

```
<?php
// $mysqli = new mysqli("servername", "username", "password", "dbname");
$mysqli = new mysqli("localhost", "root", "", "etudiants");
if($mysqli->connect_error) {
    exit('Could not connect');

}

$sql = "SELECT id,prenom, age
FROM etudiants WHERE id=?";

$stmt = $mysqli->prepare($sql);
$stmt->bind_param("s", $_GET['q']);
$stmt->execute();
$stmt->store_result();
$stmt->bind_result($id, $prenom, $age);
$stmt->fetch();
$stmt->close();

echo "<table>";
echo "<tr>";
echo "<th>ID</th>";
echo "<td>" . $id . "</td>";
echo "<th>Prenom</th>";
echo "<td>" . $prenom . "</td>";
echo "<th>Age</th>";
echo "<td>" . $age . "</td>";
echo "</tr>";
echo "</table>";
?>
```

- ▶ nous allons utiliser les fichiers **usa.xml** et **usa.json** .

A. JSON

- Chargez le fichier **usa.json** à l'aide d'une requête **AJAX** et affectez-le à la variable **usa**
- Remplir la liste **#la** avec les noms des états (State) de US.
- Attachez à la liste **#la** un handler **change**. Le handler doit synchroniser les villes de la liste **#lb** avec l'état de la liste **#la**.

B. XML

- Refaire les questions précédentes avec le fichier **usa.xml**

[Telecharger les fichiers ici :](#)
[usa.json](#) & [usa.xml](#)

1. Refaire les questions précédentes avec le fichier **usa.xml**

Afficher ou Masquer l'énoncé.

```
<div id="ce">
    <select id="la">
        <option value="">Sélectionner un état</option>
    </select>
    <select id="lb">
        <option value="">Sélectionner une ville</option>
    </select>
</div>
```

Résultat

Sélectionner un état ▾ Sélectionner une ville ▾

Partie 12 :



Qu'est-ce que jQuery ?

341

- ▶ jQuery est une bibliothèque JavaScript rapide, petite et riche en fonctionnalités. Il rend les choses comme la traversée et la manipulation de documents HTML, la gestion des événements, l'animation et Ajax beaucoup plus simples avec une API facile à utiliser qui fonctionne sur une multitude de navigateurs. Avec une combinaison de polyvalence et d'extensibilité, jQuery a changé la façon dont des millions de personnes écrivent JavaScript.
- ▶ Le but de jQuery est de faciliter l'utilisation de JavaScript sur votre site Web.
- ▶ La bibliothèque jQuery contient les fonctionnalités suivantes :
 - Manipulation HTML/DOM
 - Manipulation CSS
 - Méthodes d'événement HTML
 - Effets et animations
 - AJAX
 - Utilitaires
- ▶ **De plus, jQuery a des plugins pour presque toutes les tâches.**
- ▶ Il existe de nombreuses autres bibliothèques JavaScript, mais jQuery est probablement la plus populaire et la plus extensible.
- ▶ des plus grandes entreprises du Web utilisent jQuery, telles que :Google Microsoft IBM Netflix.
- ▶ jQuery fonctionnera exactement de la même manière dans tous les principaux navigateurs.

Ajout de jQuery à vos pages Web

342

- ▶ Il existe plusieurs façons de commencer à utiliser jQuery sur votre site Web. Tu peux: Téléchargez la bibliothèque jQuery sur [jQuery.com](https://jquery.com/) Inclure jQuery à partir d'un CDN.
- ▶ Télécharger jquery: <https://code.jquery.com/jquery-3.6.0.min.js>
- ▶ Placez le fichier téléchargé dans le même répertoire que les pages où vous souhaitez l'utiliser. La bibliothèque jQuery est un fichier JavaScript unique, et vous le référez avec la `<script>`balise HTML
(notez que la `<script>`balise doit être à l'intérieur de la `<head>`section) :`<script src="jquery-3.6.0.min.js"></script>`
- ▶ Si vous ne souhaitez pas télécharger et héberger vous-même jQuery, vous pouvez l'inclure à partir d'un CDN (Content Delivery Network).
- ▶ Google est un exemple de quelqu'un qui héberge jQuery :
- ▶ `<script src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>`
- ▶ Pour inclure les CDNs : <https://cdnjs.com/libraries/jquery>
- ▶ Si votre site Web contient de nombreuses pages et que vous souhaitez que vos fonctions jQuery soient faciles à entretenir, vous pouvez placer vos fonctions jQuery dans un fichier .js séparé.

Syntaxe Jquery

343

- ▶ Avec jQuery, vous sélectionnez (interrogez) des éléments HTML et effectuez des "actions" sur eux.
- ▶ La syntaxe jQuery est conçue sur mesure pour **sélectionner** des éléments HTML et effectuer une **action** sur le ou les éléments.
- ▶ La syntaxe de base est : **`$(selector).geste ()`**
 - Un signe \$ pour définir/accéder à jQuery
 - Un (*sélecteur*) pour "interroger (ou rechercher)" des éléments HTML
 - *Une action* jQuery () à effectuer sur le ou les élément(s)

L'événement Document prêt

```
$(document).ready(function(){
    // jQuery methods go here...
});
```

Exemples:

`$(this).hide()` - masque l'élément courant.

`$("p").hide()` - cache tous les éléments <p>.

`$(".test").hide()` - cache tous les éléments avec class="test".

`$("#test").hide()` - cache l'élément avec id="test".

jQuery a également créé une méthode encore plus courte pour l'événement document ready :

```
$(function(){
    // jQuery methods go here...
});
```

Sélecteurs jQuery

344

- ▶ Les sélecteurs jQuery sont utilisés pour "trouver" (ou sélectionner) des éléments HTML en fonction de leur nom, identifiant, classes, types, attributs, valeurs d'attributs et bien plus encore. Il est basé sur les sélecteurs CSS existants et, en plus, il possède ses propres sélecteurs personnalisés.
- ▶ Tous les sélecteurs dans jQuery commencent par le signe dollar et les parenthèses : `$()`.
- ▶ Le sélecteur d'éléments jQuery sélectionne les éléments en fonction du nom de l'élément. Vous pouvez sélectionner tous les `<p>`éléments d'une page comme ceci :`$("p")`

```
<script src="jquery-3.6.0.min.js"></script>
<script>
$(document).ready(function(){
    $("button").click(function(){
        $("p").hide();
    });
});

</script>
</head>
<body>
    <p>Ce texte ce cache lorsqu'on clique</p>
    <button>Cacher</button>
</body>
</html>
```

Ce texte ce cache lorsqu'on clique

Cacher

Sélecteurs #id /.class

345

Le sélecteur jQuery utilise l'attribut id d'une balise HTML pour trouver l'élément spécifique.**#id**

Un identifiant doit être unique dans une page, vous devez donc utiliser le sélecteur #id lorsque vous souhaitez rechercher un élément unique et unique.

```
$("#test")
```

Exemple

Lorsqu'un utilisateur clique sur un bouton, l'élément avec id="test" sera masqué :

```
$(document).ready(function(){
  $("button").click(function(){
    $("#test").hide();
  });
});|
```

Le sélecteur .class

.class Le sélecteur jQuery trouve des éléments avec une classe spécifique.

Pour rechercher des éléments avec une classe spécifique, écrivez un point suivi du nom de la classe :

```
$(".test")
```

```
$(document).ready(function(){
  $("button").click(function(){
    $(".test").hide();
  });
});|
```

Autres sélecteurs

346

<code>\$("")</code>	Selects all elements
<code>\$(this)</code>	Selects the current HTML element
<code>\$("p.intro")</code>	Selects all <code><p></code> elements with <code>class="intro"</code>
<code>\$("p:first")</code>	Selects the first <code><p></code> element
<code>\$("ul li:first")</code>	Selects the first <code></code> element of the first <code></code>
<code>\$("ul li:first-child")</code>	Selects the first <code></code> element of every <code></code>
<code>\$("[href]")</code>	Selects all elements with an <code>href</code> attribute
<code>\$("a[target='_blank']")</code>	Selects all <code><a></code> elements with a <code>target</code> attribute value equal to <code>"_blank"</code>
<code>\$("a[target!='_blank']")</code>	Selects all <code><a></code> elements with a <code>target</code> attribute value NOT equal to <code>"_blank"</code>
<code>\$(":button")</code>	Selects all <code><button></code> elements and <code><input></code> elements of <code>type="button"</code>
<code>\$("tr:even")</code>	Selects all even <code><tr></code> elements
<code>\$("tr:odd")</code>	Selects all odd <code><tr></code> elements

Utilisez [jQuery Selector Tester](#) pour démontrer les différents sélecteurs. Pour une référence complète de tous les sélecteurs jQuery, consulter [Référence des sélecteurs jQuery](#).

Méthodes d'événement jQuery

347

- ▶ Toutes les différentes actions des visiteurs auxquelles une page Web peut répondre sont appelées événements.
- ▶ Un événement représente le moment précis où quelque chose se produit.
- ▶ Exemples:
 - déplacer une souris sur un élément
 - sélection d'un bouton radio
 - cliquer sur un élément

Pour attribuer un événement de clic à tous les paragraphes d'une page, vous pouvez procéder comme suit :

```
$( "p" ).click();
```

Mouse Events	Keyboard Events	Form Events	Document/Window Events
click	keypress	submit	load
dblclick	keydown	change	resize
mouseenter	keyup	focus	scroll
mouseleave		blur	unload

L'étape suivante consiste à définir ce qui doit se passer lorsque l'événement se déclenche. Vous devez passer une fonction à l'événement :

```
$( "p" ).click(function()
{
  // action ici!!
});
```

`$(document).ready()` méthode nous permet d'exécuter une fonction lorsque le document est entièrement chargé ,elle inclue tous le code:

```
$( "p" ).dblclick(function(){
  $(this).hide();
});
```

```
$( "p" ).click(function(){
  $(this).hide();
});
```

Méthodes d'événement jQuery

348

mouseenter(): La fonction est exécutée lorsque le pointeur de la souris entre dans l'élément HTML :

```
$("#p1").mouseenter(function(){
    alert("p1!");
});
```

mouseleave(): La fonction est exécutée lorsque le pointeur de la souris quitte l'élément HTML :

```
$("#p1").mouseleave(function(){
    alert("Bye! Bye!");
});
```

mousedown() :La fonction est exécutée, lorsque le bouton gauche, central ou droit de la souris est enfoncé, alors que la souris est sur l'élément HTML :

```
$("#p1").mousedown(function(){
    alert("Mouse down!");
});
```

mouseup() La fonction est exécutée, lorsque le bouton gauche, central ou droit de la souris est relâché, alors que la souris est sur l'élément HTML :

```
$("#p1").mouseup(function(){
    alert("Mouse up!");
});
```

hover(): méthode prend deux fonctions et est une combinaison des méthodes **mouseenter()** et **mouseleave()**

```
$("#p1").hover(function(){
    alert("You entered p1!");
},function(){alert("Bye! You now leave p1!");});
```

Méthodes d'événement jQuery

349

focus() La fonction est exécutée lorsque le champ de formulaire obtient le focus :

```
$( "input" ).focus( function(){$(this).css("background-color", "#cccccc");});
```

blur() La fonction est exécutée lorsque le champ de formulaire perd le focus :

```
$( "input" ).blur( function(){$(this).css("background-color", "#ffffff");});
```

on() méthode attache un ou plusieurs gestionnaires d'événements pour les éléments sélectionnés.

```
$( "p" ).on("click", function(){$(this).hide();});
```

L'événement **submit** se produit lorsqu'un formulaire est soumis.

```
$(document).ready(function(){$('form').submit(function(){alert("Submitted");});});
```

Pour une référence complète des événements jQuery, veuillez consulter la [Référence des événements jQuery](#).

Exemple évènement avec la méthode on

350

```
$(document).ready(function(){
  $("p").on({
    mouseenter: function(){
      $(this).css("background-color",
"lightgray");
    },
    mouseleave: function(){
      $(this).css("background-color",
"lightblue");
    },
    click: function(){
      $(this).css("background-color", "yellow");
    }
  });
});
```

Effets jQuery

351

Avec jQuery, vous pouvez masquer et afficher des éléments HTML avec les méthodes `hide()` et `:show()`

```
$(document).ready(function(){
    $("button").click(function(){
        $("p").hide(1000); //il disparait après 1s
    });
});
```

On bascule entre le masquage et l'affichage d'un élément avec la méthode `toggle()`.

```
$("#hide").click(function(){
    $("p").hide();
});
$("#show").click(function(){
    $("p").show();
});
```

```
$(document).ready(function(){
    $("button").click(function(){
        $("p").toggle(); //toggle('fast') ou slow
    });
});
```

masquer un élément dans et hors de la visibilité.
jQuery a les méthodes de fondu suivantes :

- `fadeIn()`
- `fadeOut()`
- `fadeToggle()`
- `fadeTo()`

La méthode `fadeTo()` permet un fondu à une opacité donnée (valeur entre 0 et 1).

```
$("button").click(function(){
    $("#div1").fadeIn();
    $("#div2").fadeIn("slow");
    $("#div3").fadeIn(3000);
});
```

```
$("button").click(function(){
    $("#div1").fadeTo("slow", 0.15);
    $("#div2").fadeTo("slow", 0.4);
    $("#div3").fadeTo("slow", 0.7);
});
```

Effets Jquery

352

Avec jQuery, vous pouvez créer un effet de glissement sur les éléments. jQuery a les méthodes de slide suivantes :

- `slideDown()`
- `slideUp()`
- `slideToggle()`

```
$(document).ready(function(){
  $("#flip").click(function(){
    $("#panel").slideDown("slow");
  });
});
```

La méthode animate()

La méthode jQuery `animate()` est utilisée pour créer des animations personnalisées.

```
$(document).ready(function(){
  $("button").click(function(){
    $("div").animate({left: '250px'});
  });
});
```

Il déplace un élément `<div>` vers la droite, jusqu'à ce qu'il ait atteint une propriété `left` de 250px :

```
$("button").click(function()
{$("div").animate(
{left: '250px', opacity:'0.5',height: '150px',width: '150px'});});
```



Animation

353

Vous pouvez spécifier la valeur d'animation d'une propriété sous la forme " **show**", " **hide**" ou " **toggle**" :

```
$( "button" ).click(function(){
    $( "div" ).animate({
        height: 'toggle' }));});
```

vous pouvez écrirez plusieurs **animate()** appels les uns après les autres,
jQuery crée une file d'attente "interne" avec ces appels de méthode.
Ensuite, il exécute les appels animés UN par UN.

La méthode jQuery stop() est utilisée pour arrêter une animation ou un effet avant qu'il ne soit terminé.

```
$( "#stop" ).click(function(){
    $( "#panel" ).stop();
});
```

```
$( "button" ).click(function(){
    var div = $( "div" );
    div.animate({height: '300px',
    opacity: '0.4'}, "slow");
    div.animate({width: '300px',
    opacity: '0.8'}, "slow");
    div.animate({height: '100px',
    opacity: '0.4'}, "slow");
    div.animate({width: '100px',
    opacity: '0.8'}, "slow");
});
```

Manipulation DOM avec jQuery

mohamed@goumih.com

354

Trois méthodes jQuery simples mais utiles pour la manipulation du DOM sont :

- `text()`- Définit ou renvoie le contenu textuel des éléments sélectionnés
- `html()`- Définit ou renvoie le contenu des éléments sélectionnés (y compris le balisage HTML)
- `val()`- Définit ou renvoie la valeur des champs de formulaire

```
$("#btn1").click(function(){
  alert("Value: " +
 $("#test").val());
});
```

```
$("#btn1").click(function(){
  alert("Text: " + $("#test").text());
});
$("#btn2").click(function(){
  alert("HTML: " + $("#test").html());
});
```

```
$("#btn1").click(function(){
  $("#test1").text(function(i, origText){
    return "Old text: " + origText + " New
text: Hello world!
(index: " + i + ")");
});});

$("#btn2").click(function(){
  $("#test2").html(function(i, origText){
    return "Old html: " + origText + " New
html: Hello <b>world!</b>
(index: " + i + ")");
});});
```

La méthode jQuery `attr()` est utilisée pour obtenir les valeurs des attributs.

```
$(button).click(function(){
  $("#dev").attr("href", "https://
www.devdigital.com");
});
```

```
$(button).click(function()
{
  alert($("#dev").attr("href
")));
});
```

Ajouter un nouveau contenu HTML

355

Nous examinerons quatre méthodes jQuery utilisées pour ajouter du nouveau contenu :

- **append()** - Insère du contenu à la fin des éléments sélectionnés
- **prepend()** - Insère du contenu au début des éléments sélectionnés
- **after()** - Insère du contenu après les éléments sélectionnés
- **before()** - Insère le contenu avant les éléments sélectionnés

```
function afterText() {
    var txt1= "<b>I</b>";
    var txt2 = $("<i></i>").text("practice");
    var txt3 =document.createElement("b");
    txt3.innerHTML = "jQuery!";
    $("img").after(txt1, txt2,
    txt3); }
```

```
$( "p" ).prepend("Texte ajoutee.");
```

```
function appendText() {
    var txt1 = "<p>Text.</p>";
    var txt2 = $("<p></p>").text("Text.");
    var txt3 =document.createElement("p");
    txt3.innerHTML = "Text.";
    $("body").append(txt1, txt2, txt3);
}
```

Pour supprimer des éléments et du contenu, il existe principalement deux méthodes jQuery :

- **remove()**- Supprime l'élément sélectionné (et ses éléments enfants)
- **empty()**- Supprime les éléments enfants de l'élément sélectionné

```
$("#div1").remove();
$("#div1").empty();
```

CSS avec Jquery

356

jQuery a plusieurs méthodes pour la manipulation CSS. Nous allons voir les méthodes suivantes :

- **addClass()**- Ajoute une ou plusieurs classes aux éléments sélectionnés
- **removeClass()**- Supprime une ou plusieurs classes des éléments sélectionnés
- **toggleClass()**- Bascule entre l'ajout/la suppression de classes des éléments sélectionnés
- **css()**- Définit ou renvoie l'attribut de style

```
$("button").click(function(){
    $("h1, h2, p").addClass("blue");
    $("div").addClass("important");
});
```

//Spécifier plusieurs classes

```
$("button").click(function(){
    $("h1, h2, p").removeClass("blue");
});
```

//Supprimer classe avec removeClass

```
$("button").click(function(){
    $("#div1").addClass("important blue");
});
```

//toggleClass

```
$("button").click(function(){
    $("h1, h2, p").toggleClass("blue");
});
```

css() méthode définit ou renvoie une ou plusieurs propriétés de style

```
$("p").css("background-color", "yellow"); $("p").css("background-color");
```

```
$("p").css({"background-color": "yellow", "font-size": "200%"});
```

Pour un aperçu complet de toutes les méthodes CSS jQuery, veuillez consulter la [référence jQuery HTML/CSS](#).

Méthodes de dimension jQuery

357

jQuery propose plusieurs méthodes importantes pour travailler avec les dimensions :

- **width()** définit ou renvoie la largeur d'un élément (exclut le rembourrage, la bordure et la marge).
- **height()** méthode définit ou renvoie la hauteur d'un élément (exclut le rembourrage, la bordure et la marge).
- **innerWidth()** renvoie la largeur d'un élément (comprend le rembourrage).
- **innerHeight()** renvoie la hauteur d'un élément (y compris le rembourrage).
- **outerWidth()** renvoie la largeur d'un élément (y compris le rembourrage et la bordure).
- **outerHeight()** renvoie la hauteur d'un élément (y compris le rembourrage et la bordure).

```
$("button").click(function(){
    var txt = "";
    txt += "Outer width: " +
    $("#div1").outerWidth()
    + "<br>";
    txt += "Outer height: " +
    $("#div1").outerHeight();
    $("#div1").html(txt);
});
```

```
$("button").click(function(){
    var txt = "";
    txt += "Inner width: " +
    $("#div1").innerWidth()
    + "<br>";
    txt += "Inner height: " +
    $("#div1").innerHeight();
    $("#div1").html(txt);
});
```

```
$("button").click(function(){
    var txt = "";
    txt += "Document width/height: " +
    $(document).width();
    txt += "x" + $(document).height()
    + "\n";
    txt += "Window width/height: " +
    $(window).width();
    txt += "x" + $(window).height();
    alert(txt);
});
```

le Traversage

358

traversée jQuery, qui signifie "se déplacer", est utilisée pour "trouver" (ou sélectionner) des éléments HTML en fonction de leur relation avec d'autres éléments. Commencez par une sélection et parcourez cette sélection jusqu'à ce que vous atteigniez les éléments souhaités.

```
$(document).ready(function(){
    $("h1").add("p").add("span").css("background-color", "yellow");
});
```

Méthodes de parcours jQuery:

https://www.w3schools.com/jquery/jquery_ref_traversing.asp

Trois méthodes jQuery utiles pour parcourir l'arborescence DOM sont :

- parent()**
- parents()**
- parentsUntil()**

```
$(document).ready(function(){
    $("span").parent();
});
```

```
$(document).ready(function(){
    $("span").parents();
});
```

La **parentsUntil()** méthode renvoie tous les éléments ancêtres entre deux arguments donnés.

```
$(document).ready(function(){
    $("span").parentsUntil("div");
});
```

jQuery Traversing - Descendants

359

`children()` méthode renvoie tous les enfants directs de l'élément sélectionné.

Cette méthode ne traverse qu'un seul niveau dans l'arborescence DOM.

`find()`: méthode renvoie les éléments descendants de l'élément sélectionné, jusqu'au dernier descendant.

```
$(document).ready(function(){
  $("div").find("span");
});
```

```
$(document).ready(function(){
  $("div").find("*");
});
```

Il existe de nombreuses méthodes jQuery utiles pour parcourir latéralement l'arborescence DOM :

- `siblings()`: renvoie tous les éléments frères de l'élément sélectionné.
- `next()` : renvoie l'élément frère suivant de l'élément sélectionné.
- `nextAll()`: renvoie tous les éléments frères suivants de l'élément sélectionné.
- `nextUntil()`: renvoie tous les éléments frères suivants entre deux arguments donnés.

```
$(document).ready(function(){
  $("div").children("p.first");
});
```

```
$(document).ready(function(){
  $("div").children();
});
```

```
$(document).ready(function(
){
  $("h2").nextAll();
});
```

```
$(document).ready(function(
){
  $("h2").siblings();
});
```

Les méthodes `prev()`, `prevAll()` et `prevUntil()` fonctionnent exactement comme les méthodes ci-dessus mais avec une fonctionnalité inverse : elles renvoient les éléments frères précédents

jQuery Traversing - Filtrage

360

Les méthodes **first()**, **last()** permettent de sélectionner permettre de sélectionner le premier et le dernier élément.

```
$(document).ready(function(){
    $("div").first();});
```

La méthode **filter()** vous permet de spécifier un critère. Les éléments qui ne correspondent pas aux critères sont supprimés de la sélection et ceux qui correspondent sont renvoyés. L'exemple suivant renvoie tous les éléments avec le nom de classe "intro":

```
$(document).ready(function(){
    $("p").filter(".intro");
});
```

```
$(document).ready(function()
{
    $("div").last();});
```

La méthode **eq()** renvoie un élément avec un numéro d'index spécifique des éléments sélectionnés. Les numéros d'index commencent à 0, donc le premier élément aura le numéro d'index 0 et 2eme 1

```
$(document).ready(function(){
    $("p").eq(1);
});
```

La méthode **not()** renvoie tous les éléments qui ne correspondent pas aux critères. Astuce : La méthode **not()** est l'opposé de **filter()**. L'exemple suivant renvoie tous les éléments qui n'ont pas le nom de classe "intro":

```
$(document).ready(function(){
    $("p").not(".intro");
});
```

jQuery - La méthode noConflict()

361

jQuery utilise le signe \$ comme raccourci pour jQuery. Il existe de nombreux autres frameworks JavaScript populaires tels que : Angular, Backbone, Ember, Knockout, etc. Et si d'autres frameworks JavaScript utilisaient également le signe \$ comme raccourci ? Si deux frameworks différents utilisent le même raccourci, l'un d'eux peut cesser de fonctionner. L'équipe jQuery y a déjà pensé et a implémenté la méthode noConflict().

La méthode **noConflict()** libère la retenue sur l'identificateur de raccourci \$, afin que d'autres scripts puissent l'utiliser. Vous pouvez bien sûr toujours utiliser jQuery, simplement en écrivant le nom complet à la place du raccourci :

```
$ .noConflict();
jQuery(document).ready(function(){
  jQuery("button").click(function(){
    jQuery("p").text("jQuery is still
working!");
  });
});
```

Vous pouvez également créer votre propre raccourci très facilement. La méthode **noConflict()** renvoie une référence à jQuery, que vous pouvez enregistrer dans une variable, pour une utilisation ultérieure. Voici un exemple:

```
var jq = $.noConflict();
jq(document).ready(function(){
  jq("button").click(function(){
    jq("p").text("jQuery is still working!");
  });
});
```

Si vous avez un bloc de code jQuery qui utilise le raccourci \$ et que vous ne voulez pas tout changer, vous pouvez passer le signe \$ en paramètre à la méthode ready. Cela vous permet d'accéder à jQuery en utilisant \$, à l'intérieur de cette fonction - en dehors de celle-ci, vous devrez utiliser "jQuery":

```
$.noConflict();
jQuery(document).ready(function($){
  $("button").click(function(){
    $("p").text("jQuery is still working!");
  });
});
```

Ajax Jquery

mohamed@goumih.com

362

Exemple avec JQUERY :la notion de charger les donnés de l'extérieur.

jQuery - Méthode AJAX load()

La méthode jQuery `load()` est une méthode AJAX simple mais puissante.

La `load()` méthode charge les données d'un serveur et place les données renvoyées dans l'élément sélectionné.

Syntaxe:

```
$(selector).load(URL,data,callback);
```

```
$("#div1").load("demo_test.txt");
```

Il est également possible d'ajouter un sélecteur jQuery au paramètre URL.

L'exemple suivant charge le contenu de l'élément avec id="p1", à l'intérieur du fichier "demo_test.txt", dans un `<div>` élément spécifique :

```
<script>
    $("button").click(function(){
        $("#div1").load("text3.html #p1");
    });
</script>
```

Le paramètre callback facultatif spécifie une fonction de rappel à exécuter lorsque la `load()` méthode est terminée.

La fonction de rappel peut avoir différents paramètres :

- `responseTxt`- contient le contenu résultant si l'appel réussit
- `statusTxt`- contient le statut de l'appel
- `xhr`- contient l'objet XMLHttpRequest

```
$("#div1").load("test1.txt", function(responseTxt, statusTxt, xhr){
    if(statusTxt == "success")
        alert("Le document est chargé avec succès!");
    if(statusTxt == "error")
        alert("Error: " + xhr.status + ": " + xhr.statusText);
});
```

jQuery - Méthodes AJAX get() et post()

363

Les méthodes jQuery get() et post() sont utilisées pour demander des données au serveur avec une requête HTTP GET ou POST.

La `$.get()` méthode demande des données au serveur avec une requête HTTP GET.

```
$("button").click(function(){
    $.get("demo_test.php", function(data, status){
        alert("Data: " + data + "\nStatus: " + status);
    });
});
```

La `$.post()` méthode demande des données au serveur à l'aide d'une requête HTTP POST.

```
$(document).ready(function(){
    $("button").click(function(){
        $.post("post.php",
        {
            name: "mohamed",
            city: "agadir"
        },
        function(data,status){
            alert("Data: " + data + "\nStatus: " + status);
        });
    });
});
```

On considère la structure html ci-dessous. Exécuter, à l'aide jQuery les manipulations suivantes :

1. Modifier la couleur du texte de la première div (rouge)
2. Cloner la première et l'ajouter à la fin de la liste des div
3. Déplacer la dernière div et la mettre au début de la liste
4. Créer une nouvelle div, l'ajouter à la fin de la liste et afficher dedans le nombre total des div de la liste (y compris celle ajoutée)
5. Parcourir toutes les div et y ajouter un numéro d'ordre 1, 2, ...

Résultat

Ma première div 11111

Ma deuxième div 22222

Ma troisième div 33333

Ma quatrième div 44444

1. Avec jquery ,créer un script qui permet de détecter le geste de souris avec un clic sur une paragraphe :soit un seul ou double clic avec coloration du message ,afficher le résultat dans un div.
2. Avec jquery désactiver le clic sur un Botton droit dans une page html

un seul click

Click me!

double click

3.Avec Jquery ,créer un script qui permet scroller une paragraphe et aider l'utilisateur de la lire de haut de bas ou de bas de haut ,en cliquant sur des liens bas et top et,pour un moment déterminé en secondes sans bouger le cursus de la souris

4.Avec Jquery ,écrire une fonction qui permet de fadeOut/fadeIn un span dans un texte chaque 1s.

paragraphe contient un span qui fadeIn/fadeOut chaque 1s

5.Avec jquery ,créer un script qui permet colorer les éléments pairs d'un table avec même couleur différente des autres .

Colorer les étudiants qui ont une note ≥ 18 avec background-color jeune.

Ajouter un Button et un input qui permet de déterminer le numéro de ligne à supprimer

Lorsqu'on clique sur le Button ,vérifier que l'utilisateur ne pouvez pas supprimer l'entete du tableau.

Ajouter 2 inputs qui permet d'ajouter un etudiant avec sa note dans la table.

Ajouter un div qui affiche la moyenne des notes à chaque fois on ajoute ou ons supprime

[Lire le texte vers me bas](#)

Lorem ipsum dolor sit amet consectetur adipisicing elit. Minima delectus quaerat maxime, quod blanditiis amet tenetur vel, quam eveniet quos distinctio impedit repellendus optio, laboriosam accusantium iusto explicabo necessitatibus! Fugit, soluta! Et, ex exercitationem! A qui repellat harum saepe nam tempore aliquid nisi excepturi, placeat dignissimos est sapiente, culpa nihil mollitia, deserunt recusandae. Numquam, dolorem minima nihil dignissimos consectetur exercitationem nesciunt voluptatum distinctio mollitia veritatis magni, eaque, est veniam ullam eum temporibus! Possimus maxime provident accusamus illum qui,

Etudiant Note

Mohamed	18
Hassan	20
Imane	14

Etudiant Note

Mohamed	18
Hassan	20
Imane	14
Hilal	10

366

1.Attachez un handler pour l'événement **keydown** sur l'objet **document**

2.Si la touche **SHIFT** est appuyée et que l'on apuie sur une touche de direction (\leftarrow , \uparrow , \rightarrow ou \downarrow) la div **div.box** se déplace dans la direction choisie.

The screenshot shows a browser's developer tools interface. The top section is a code editor with the following content:

```
<div id="ce">
  <style>
    .box {position: relative; width: 50px; height: 50px; background: red; margin: 10px auto; transform: rotate(-15deg); transition: all 0.5s ease-out; border-radius: 50%; overflow: hidden; z-index: 1;}</style>
  <div class="box"></div>
</div>
```

The bottom section is labeled "Résultat" and displays the rendered HTML. It shows a single red square element with a 15-degree clockwise rotation, centered on the page with a 10px margin from the edges. The element has rounded corners and a slight shadow.

TP12:

367

- Avec Jquery ,écrire un script qui permet au utilisateur de cliquer sur deux Buttons pour afficher le formulaire du login ou inscription

Login Inscription

Faire un menu de style qui affiche des paragraphes et les masqués selon le style des slides

Panel 1

Panel 2

Panel 3

Panel 4

Realiser l'interface suivante

My Favorite Movies

Title Rating Add Movie

Title	Rating	Delete

9.Créer l'animation suivante sur des divs



TP13: Ajax avec Jquery

368

1. Créer une balise <pre> et l'ajouter dans un div
2. Ajouter la classe CSS **pre1** à la balise <pre>
3. Créer une balise <code> et l'ajouter dans la balise <pre>
4. Faire une requête AJAX pour récupérer le fichier user.json sous format texte
5. Ajouter son contenu dans la balise <pre> ,
6. ajouter l'element **token** dans la balise code.
7. Récupérez les utilisateurs de **users.json** et stockez-les dans la variable **users**
8. Créez un tableau html
9. Ajouter une ligne avec les entetes **Email** et **Token**
10. Pour chaque utilisateur, ajouter une ligne et des cellules l'**email** et le **token**

```

user.json  X
es > {} user.json > {} 1
1  [
2  {
3    "email": "toto@example.com",
4    "token": "0D0DHG765HS5FGS5"
5  },
6  []
{} users.json  X
files > {} users.json > ...
1  [
2  {
3    "email": "toto@example.com",
4    "token": "0D0DHG765HS5FGS5"
5  },
6  {
7    "email": "reda@example.com",
8    "token": "0D0DHG765HS5FGS5"
9  },
10 {
11   "email": "momo@example.com",
12   "token": "0D0DHG765HS5FGS5"
13 }
14 ]

```

TP14:Jquery & Ajax

mohamed.goumih@ofppt.ma

369

- Opérations sur un tableau HTML :
1. Parcourir les lignes du tableau et afficher les nombres de produits leur prix ,quantité.
 2. Calculer les sous-totaux
 3. Calculer le total général

```
<td>Scanner</td>
<td class="r">2400.00</td>
<td class="r">2</td>
<td class="r">4800.00</td></tr>
<tr>
<td>onduleur</td>
<td class="r">1200.00</td>
<td class="r">3</td>
<td class="r">3600.00</td></tr>
<tr>
<td colspan="3" class="r">TOTAL : </td>
<td class="r">39600.00</td>
</tr>
...
```

Résultat

PRODUIT	PRIX	QUANTITE	SOUS-TOTAL
PC	5400.00	3	16200.00
Imprimante	1500.00	10	15000.00
Scanner	2400.00	2	4800.00
Onduleur	1200.00	3	3600.00
TOTAL :			39600.00

Partie 13 :

**Allez plus loin avec
JavaScript**

Modules

371

Un module n'est qu'un fichier. Un script est un module. Aussi simple que cela.

Les modules peuvent se charger mutuellement et utiliser des directives spéciales, **export** et **import**, pour échanger des fonctionnalités,

appeler les fonctions d'un module dans un autre:

- Le mot-clé **export** labelise les variables et les fonctions qui doivent être accessibles depuis l'extérieur du module actuel.
- **import** permet l'importation de fonctionnalités à partir d'autres modules.

Par exemple, si nous avons un fichier **sayHi.js** exportant une fonction :

```
// sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
```

Un autre fichier peut l'importer et l'utiliser:

```
// main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

Comme les modules prennent en charge des mots-clés et des fonctionnalités spéciales, nous devons indiquer au navigateur qu'un script doit être traité comme un module, en utilisant l'attribut `<script type="module">`.

Les modules fonctionnent toujours en mode strict. Par exemple. l'affectation à une variable non déclarée donnera une erreur.

Exporter et Importer

mohamed@goumih.com

372

Nous pouvons étiqueter n'importe quelle déclaration comme exportée en plaçant `export` devant elle, que ce soit une variable, une fonction ou une classe.
Par exemple, ici toutes les exports sont valides:

l'`export` avant une classe ou une fonction n'en fait pas une **function expression**. C'est toujours une fonction déclaration, bien qu'elle soit exportée.

La plupart des guides de bonnes pratiques JavaScript ne recommandent pas les points-virgules après les déclarations de fonctions et de classes.

C'est pourquoi il n'est pas nécessaire d'utiliser un point-virgule à la fin de `export class` et de `export function`:

s'il y a beaucoup à importer, nous pouvons tout importer en tant qu'objet en utilisant `import * as <obj>`, par exemple:

```
// └── main.js
import * as say from './say.js';

say.sayHi('John');
say.sayBye('John');
```

```
// exporter un tableau
export let months = ['Jan', 'Feb', 'Mar', 'Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// exporter une constante
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// exporter une classe
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

En outre, nous pouvons mettre l'export séparément. Ici, nous déclarons d'abord, puis exportons

```
// └── say.js
function sayHi(user) {
  alert(`Hello, ${user}!`);
}

function sayBye(user) {
  alert(`Bye, ${user}!`);
}

export {sayHi, sayBye}; // une liste de variables exportées
```

Axios

373

Axios est une bibliothèque HTTP client populaire pour JavaScript. Il permet de faire des requêtes HTTP à un serveur et de gérer les réponses de manière structurée et efficace.

Installation: Pour commencer, vous devez installer Axios à l'aide de npm :

npm install axios

```
// import Axios
import axios from "axios";

// Faire une demande
axios.get('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.log(error);
});

```

```
$ node axios.js
{ userId: 1, id: 1, title: 'delectus aut autem', completed: false }
```

Axios

374

POST Request

Vous pouvez faire une requête POST avec Axios pour envoyer des données à un serveur. Voici un exemple :

```
axios.post('https://jsonplaceholder.typicode.com/posts', {
  title: 'Mon nouveau post',
  body: 'Contenu de mon post',
  userId: 1
})
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.log(error);
  });

```

```
$ node axios.js
  title: 'Mon nouveau post',
  body: 'Contenu de mon post',
  userId: 1,
  id: 101
}
```

Dans cet exemple, nous envoyons un objet avec les données que nous voulons envoyer au serveur. Nous utilisons la méthode **post()** d'Axios pour envoyer la requête.

Allez plus loin

375

- Donc ce cours on a traiter à peut près 95% du cours javascript ,contenant plus du programme de la formation.
- Il reste encore des parties à apprendre pour aller plus loin avec javascript qui vont aider à manipuler l'animation avancée et les Frameworks coté client et coté serveur. Comme par exemple :
- Frames et windows
- WebSocket
- Manipulation des fichiers et URLs
- JavaScript animations
- Web components
- Pour cela il faut consulté la documentation officielle:
- <https://developer.mozilla.org/fr/docs/Web/JavaScript>
- <https://devdocs.io/javascript/>
- <https://Javascrip.info>