

javascript



diginamic
FORMATION

- [1 Références](#)
- [2 Historique](#)
- [3 JavaScript Engine](#)
 - [3.1 JavaScript Runtime](#)
- [4 Commentaires](#)
- [5 Variables](#)
 - [5.1 Nom des variables ou identifiant](#)
 - [5.2 Typage dynamique](#)
 - [5.2.1.1 Selon vous, qu'affichera le code ci-dessus dans la console ?](#)
 - [5.3 Transtypage](#)
 - [5.4 Portée des variables](#)
 - [5.4.1.1 D'après vous que va afficher dans la console le code ci-dessus ?](#)

- 5.5 Const
- 6 Instructions, expressions et structures de contrôle
 - 6.1 Instructions
 - 6.1.1 Les instructions de contrôle du flux
 - 6.1.1.1 `Condition if ...else`
 - 6.1.2 `Bloc`
 - 6.1.3 `break`
 - 6.1.4 **`continue`**
 - 6.1.5 `switch`
 - 6.1.6 `throw`
 - 6.1.7 `try...catch`
 - 6.1.8 Déclarations
 - 6.1.8.1.1 `var`
 - 6.1.8.1.2 `let`
 - 6.1.8.1.3 `const`
 - 6.2 Fonctions et classes
 - 6.2.1.1.1 `function`
 - 6.2.1.1.2 `return`
 - 6.2.1.1.3 `class`
 - 6.2.1.2 Itérations
 - 6.2.1.2.1 `for`
 - 6.2.1.2.2 `while`
 - 6.2.1.2.3 `boucle do ... while`
 - 6.2.1.2.4 `boucle for ... in`

- [6.2.1.2.5 Parcours d'un tableau à index avec la boucle for](#)
- [6.2.1.2.6 Parcours d'un tableau à index avec la boucle for ... of](#)
- [6.3 Autres](#)
 - [6.3.1.1.1 `debugger`](#)
 - [6.3.1.1.2 `export`](#)
 - [6.3.1.1.3 `import`](#)
 - [6.3.1.1.4 `label`](#)
- [6.4 Expression](#)
- [6.5 Expression VS Statement \(instructions\)](#)
- [6.6 Zoom sur quelques opérateurs remarquables](#)
 - [6.6.1.1.1 Chaînage optionnel \(optional chaining\)_\(?.\)](#)
 - [6.6.1.1.2 Opérateur OR \(||\)](#)
 - [6.6.1.1.3 Comparaison](#)
 - [6.6.1.2 A votre avis que va afficher le code ci dessus ?](#)
- [6.7 Voir le détail des autres opérateurs :](#)
- [7 Fonctions](#)
 - [7.1 Fonction classique](#)
 - [7.2 Paramètres optionnels et valeurs par défaut](#)
 - [7.3 Opérateur Rest](#)
 - [7.4 Hoisting](#)
 - [7.5 Contexte d'exécution](#)

- [7.5.1 Closure](#)
 - [7.5.1.1 Autre exemple de closure](#)
- [7.6 Fonction anonyme immédiate](#)
 - [7.6.1.1 Exercice](#)
- [7.7 Arrow function](#)
- [7.8 Raccourci de déclaration de méthode dans un objet](#)
 - [7.8.1 First class citizen](#)
 - [7.8.2 Higher order function :](#)
 - [7.8.3 Fonction pure](#)
 - [7.8.4 setTimeout et setInterval](#)
- [8 Objet](#)
 - [8.1 Qui est "this" ?](#)
 - [8.2 Raccourci pour la création d'objets - depuis ES2015](#)
 - [8.3 Prototype](#)
 - [8.4 Reprenons l'exemple de code du début de cette page :](#)
 - [8.5 Exercice 1](#)
 - [8.5.1 Class et héritage](#)
 - [8.6 Propriétés privées avec getter et setter](#)
 - [8.7 Propriétés et méthodes de classe avec le mot clé static](#)
 - [8.8 Objet littéral](#)

- [8.9 Assignment déstructurée avec le spread operator](#)
- [8.10 Récupérer des informations sur les objets](#)
- [8.11 Connaître les noms des classes dont héritent une instance d'objet](#)
- [8.12 Exercice 2](#)
- [8.13 Exercice 3](#)
- [9 Tableaux](#)
 - [9.1 Créer un tableau et obtenir sa taille](#)
 - [9.2 Accéder \(via son index\) à un élément du tableau](#)
 - [9.3 Boucler sur un tableau](#)
 - [9.4 Ajouter à la fin du tableau](#)
 - [9.5 Supprimer le dernier élément du tableau](#)
 - [9.6 Supprimer le premier élément du tableau](#)
 - [9.7 Ajouter au début du tableau](#)
 - [9.8 Trouver l'index d'un élément dans le tableau en fonction d'une condition](#)
 - [9.9 Méthode "map"](#)
 - [9.9.1.1 où comment créer un nouveau tableau à partir d'un tableau existant selon une fonction de transformation](#)
 - [9.9.2 Méthode "filter"](#)

- 9.9.2.1 où comment créer un nouveau tableau à partir d'un tableau existant en filtrant selon une condition
 - 9.9.3 Supprimer des éléments à partir d'un index
- 9.10 Copier un tableau
- 9.11 Trier un tableau
 - 9.11.1.1 Exemple
 - 9.11.2 Assignment déstructurée avec le rest operator
 - 9.11.3 Assignment avec le spread operator
- 9.12 Exercice 1 : Classer des utilisateurs par age
- 9.13 Exercice 2 - Chaînage des méthodes
- 9.14 Exercice 3 - Méthode reduce
- 10 Asynchrone
 - 10.1 "Fil d'exécution" ou thread
- 11 Document Object Model (DOM)
 - 11.1 Références :
 - 11.2 Représentation générique du DOM
 - 11.3 Un exemple de construction du DOM par un navigateur à partir d'un fichier HTML
 - 11.4 Héritage des éléments du DOM
 - 11.5 Sélection avancée des éléments du DOM
 - 11.6 Exercice 1

- [11.7 Exercice 2](#)
- [11.8 Exercice 3](#)
- [11.9 Exercice 4](#)
 - [11.9.1 Fonction qui renvoie null en cas d'erreur](#)
- [12 Événements](#)
 - [12.1 Liste des événements](#)
 - [12.2 Exemple](#)
 - [12.3 Objet événement](#)
 - [12.3.1 Exemple de récupération de l'objet événement](#)
 - [12.3.2 Exemple de passage de 2 paramètres à la fonction qui est déclenchée au click sur h1](#)
 - [12.4 addEventListener](#)
 - [12.5 Gestion des événements clavier](#)
 - [12.5.1 Exemple](#)
 - [12.6 Exercices](#)
 - [12.6.1 Paragraphes Lorem ipsum](#)
 - [12.6.2 Liste de tâches](#)
- [13 Modules](#)
 - [13.1 Exports et imports multiples](#)
- [14 Mode strict](#)
 - [14.1.1 2.3 - Executer la requête](#)

- [14.2 Traitement du résultat d'une requête](#)
 - [14.2.1 Les propriétés importantes de XMLHttpRequest](#)
 - [14.2.1.1 XMLHttpRequest.status](#)
 - [14.2.1.2 XMLHttpRequest.statusText](#)
 - [14.2.1.3 XMLHttpRequest.responseText](#)
 - [14.2.2 Testons le résultat de notre requête](#)
- [14.3 Création d'une requête asynchrone](#)
 - [14.3.1 Instancier XMLHttpRequest](#)
 - [14.3.2 Préparons le terrain avec quelques fonctions sympathiques](#)
 - [14.3.2.1 Traiter la progression de la requête : XMLHttpRequest.onprogress\(\)](#)
 - [14.3.2.2 Traiter une erreur de la requête : XMLHttpRequest.onerror\(\)](#)
 - [14.3.2.3 Traiter le changement de statut de la requête : XMLHttpRequest.onload\(\)](#)
 - [14.3.3 Ouvrir la requête asynchrone](#)
 - [14.3.4 Lancement de la requête](#)
 - [14.3.5 Tester si la réponse est bien au format json avec l'objet JSON et la méthode parse](#)
- [14.4 Exercices pratiques](#)

- [14.4.1 Exercice 1 : Afficher le contenu d'une réponse dans un conteneur html](#)
- [14.4.2 Exercice 2 : Envoyer des données et traiter la réponse](#)
- [15 Promesse](#)
 - [15.1 Ancienne méthode : via des callback](#)
 - [15.2 Avec les promises](#)
 - [15.3 Le chaînage de promesses](#)
 - [15.4 Async et await](#)
 - [15.4.1 async](#)
 - [15.4.2 await](#)
 - [15.4.3 Exemple de code](#)
- [16 Fetch](#)
 - [16.1 Un exemple complet](#)
- [17 Bonnes pratiques](#)
 - [17.1 Linter](#)
 - [17.2 Imperative vs functional programming](#)
 - [17.2.1.1 Programmation fonctionnelle](#)
 - [17.2.1.2 Programmation impérative](#)
 - [17.2.1.3 Langages impératifs et processeurs](#)
 - [17.2.1.4 Instructions de la base impérative](#)

1 Références

- [La doc en français de Mozilla](#)

- [vidéo en français par Thierry Chatel, membre de Crealead](#)
- [Une introduction très complète sur le site W3Schools.com](#)
- [Le support de ce cours en pdf \(pas ou peu mis en page\)](#)

2 Historique



En 1995, **Brendan Eich**, programmeur chez Netscape développe en quelques jours un langage de script. Il fut d'abord appelé Mocha puis LiveScript puis enfin JavaScript pour surfer sur la vague du langage Java lancé à grands renforts de marketing par Sun Microsystems.

3 JavaScript Engine

JavaScript Engine ou le "moteur JavaScript" est un **programme logiciel qui interprète et exécute du code en langage JavaScript**. Les moteurs JavaScript sont généralement intégrés aux navigateurs Web mais ils peuvent

également s'exécuter dans des environnement serveur comme Node.

Le premier moteur JavaScript, SpiderMonkey, a été créé par l'informaticien américain **Brendan Eich** pour le navigateur Netscape Navigator. Il était programmé en langage C.

Les premiers moteurs JavaScript étaient de simples interpréteurs. Puis, les versions ont évolué et ont intégré la compilation afin d'améliorer les performances d'exécution du code JavaScript.

V8 est le moteur JavaScript open source le plus utilisé et le plus connu. Il est développé par Google, utilisé sur Google Chrome, Chromium et Node.js.

Cf https://fr.wikipedia.org/wiki/Moteur_JavaScript

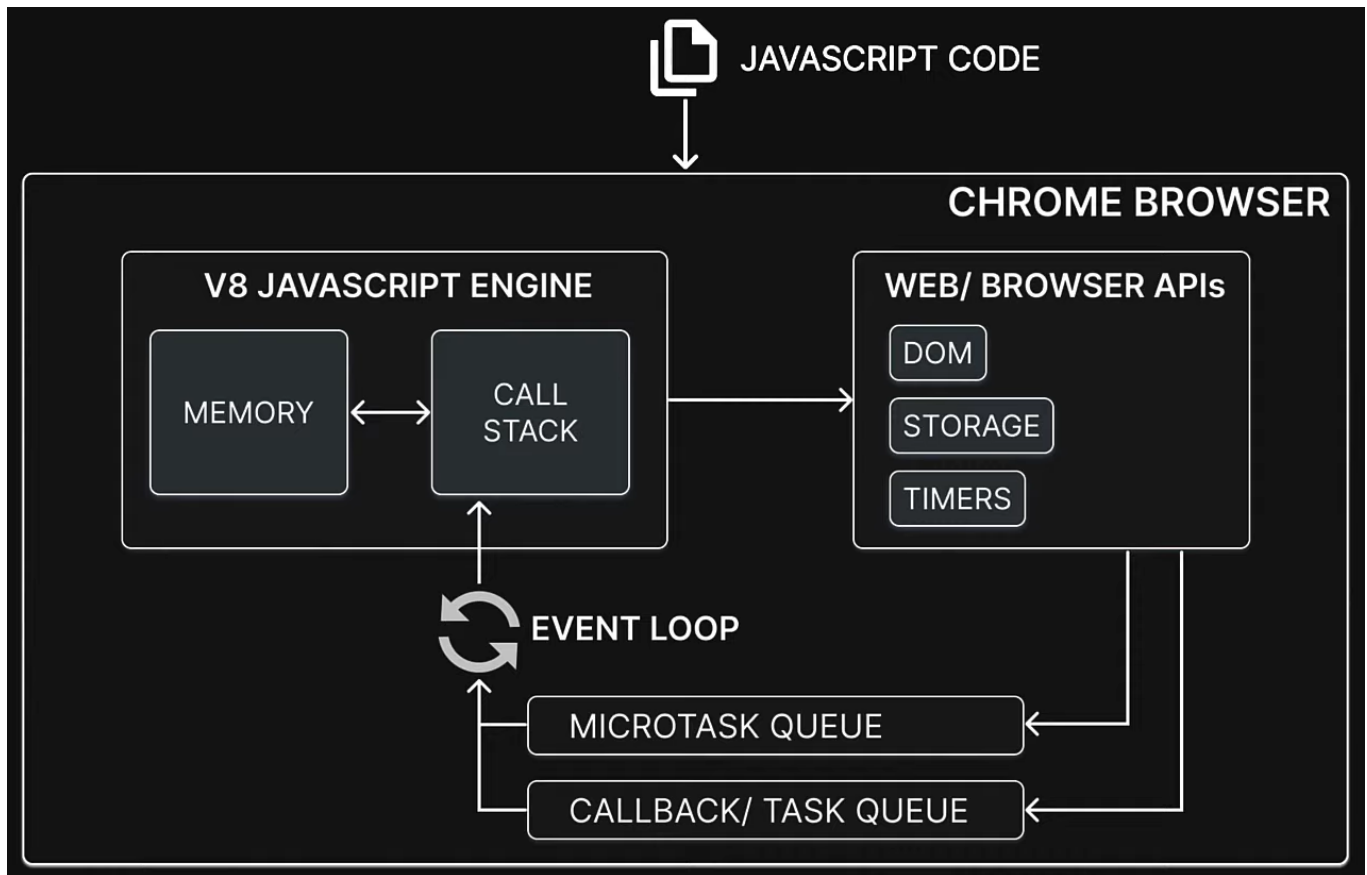
3.1 JavaScript Runtime

JavaScript Runtime est l'environnement qui fournit tous les composants nécessaires pour utiliser et exécuter des programmes js.

Prenons l'exemple du navigateur Chrome. Le schéma suivant indique que Chrome contient non-seulement un moteur de JavaScript (V8 en l'occurrence) mais aussi :

- des web APIs comme le DOM, le Storage, les Timers

- des queues (de task ou de microtasks, ces dernières ayant la priorité)
- des boucles d'événements



4 Commentaires

- <https://jsdoc.app/index.html>
- function : <https://jsdoc.app/about-getting-started.html>
- classes : <https://jsdoc.app/howto-es2015-classes.html>
- return : <https://jsdoc.app/tags-returns.html>
- async : <https://jsdoc.app/tags-async.html>

5 Variables

Si l'on en croit wikipedia, les variables sont des symboles qui associent un nom (l'identifiant) à une valeur. Le nom est unique ([mais différents des mots réservés](#)).

Dans la plupart des langages et notamment les plus courants comme javascript, les variables peuvent changer de valeur au cours du temps.

Une variable est un espace de stockage pour un résultat. Cependant les possibilités d'une variable sont intimement liées au langage de programmation auquel on fait référence. Par exemple une variable en javascript aura 4 caractéristiques :

- son **nom** c'est-à-dire sous quel nom est déclarée la variable ;
- son **type**, c'est la convention d'interprétation de la séquence de bits qui constitue la variable.
- sa **valeur**, c'est la séquence de bits elle-même
- sa **portée**, c'est la portion de code source où elle est accessible, par exemple :
 - la portée d'une variable déclarée dans une fonction avec le mot clé "**var**" (non globale) s'entend de sa définition à la fin du bloc de la dite fonction. On dit alors que la variable est "function scope"

- la portée d'une variable déclarée dans un bloc de code avec le mot clé **"let"** (non globale) s'entend de sa définition à la fin du bloc où elle est définie. On dit alors que la variable est "block scope"

5.1 Nom des variables ou identifiant

- Début du nom de la variable : Les noms des variables peuvent commencer par n'importe quel lettre [unicode](#) ou underscore ("_") ou dollar (\$). Attention à ne pas utiliser de nombre ou le "-" qui serait compris comme l'opérateur arithmétique "moins".
- Les caractères suivants doivent être des lettres unicode ou underscore ("_") ou dollar (\$)

Javascript dispose de types de données primitifs :

- string (chaînes de caractères)
- number (nombres)
- boolean (booléen)
- undefined (indéfini)
- null (nul)

... et de types de données plus évolués :

- object (objet),
- function (fonction)
- array (tableau),

- ...

5.2 Typage dynamique

Il n'est pas nécessaire de déclarer le type d'une variable avant de l'utiliser. Le type de la variable sera automatiquement déterminé lorsque le programme sera exécuté. Cela signifie également que la même variable pourra avoir différents types au cours de son existence.

L'opérateur `typeof()` permet de connaître le type de données d'une variables. Ex :

```
var i = 12;  
i = i + 3; // addition  
var j = true;  
var k = "Hellow World";  
var l;  
  
console.log(typeof(i));  
console.log(typeof(j));  
console.log(typeof(k));  
console.log(typeof(l));
```

5.2.1.1 Selon vous, qu'affichera le code ci-dessus dans la console ?

5.3 Transtypage

Il est parfois nécessaire de spécifier le type d'une variable.

On peut alors utiliser :

- parseInt
- parseFloat (ou + Ex : const customerId = + req.params.id;)
- toString

Mais il existe aussi des opérateurs :

- • pour convertir en number. [En savoir plus](#)
- !! pour convertir en booléen :

value		!!value
false		false
true		true
null		false
undefined		false
0		false
-0		false
1		true
-5		true
NaN		false

<code>''</code>		<code>false</code>
<code>'hello'</code>		<code>true</code>

5.4 Portée des variables

Avec le mot clé **"var"**, les variables sont "function scope", c'est à dire qu'elles sont définies à l'intérieur du bloc de code ({Ceci est un bloc de code}) de la fonction et inconnues en dehors.

Avec les mots clés **"let"** et **"const"**, les variables sont "block scope", c'est à dire qu'elles sont définies à l'intérieur du bloc de code ({Ceci est un bloc de code}) où elles ont été créées et inconnues au dehors.

Exercice

Examinez le code suivant :

```
{  
var i = 5;  
let j = 12;  
console.log("valeur de i dans le bloc : " + i);  
console.log("valeur de j dans le bloc : " + j);  
}  
console.log("valeur de i dans le contexte d'exécution global :  
" + i);
```

```
console.log("valeur de j dans le contexte d'exécution global :  
" + j);
```

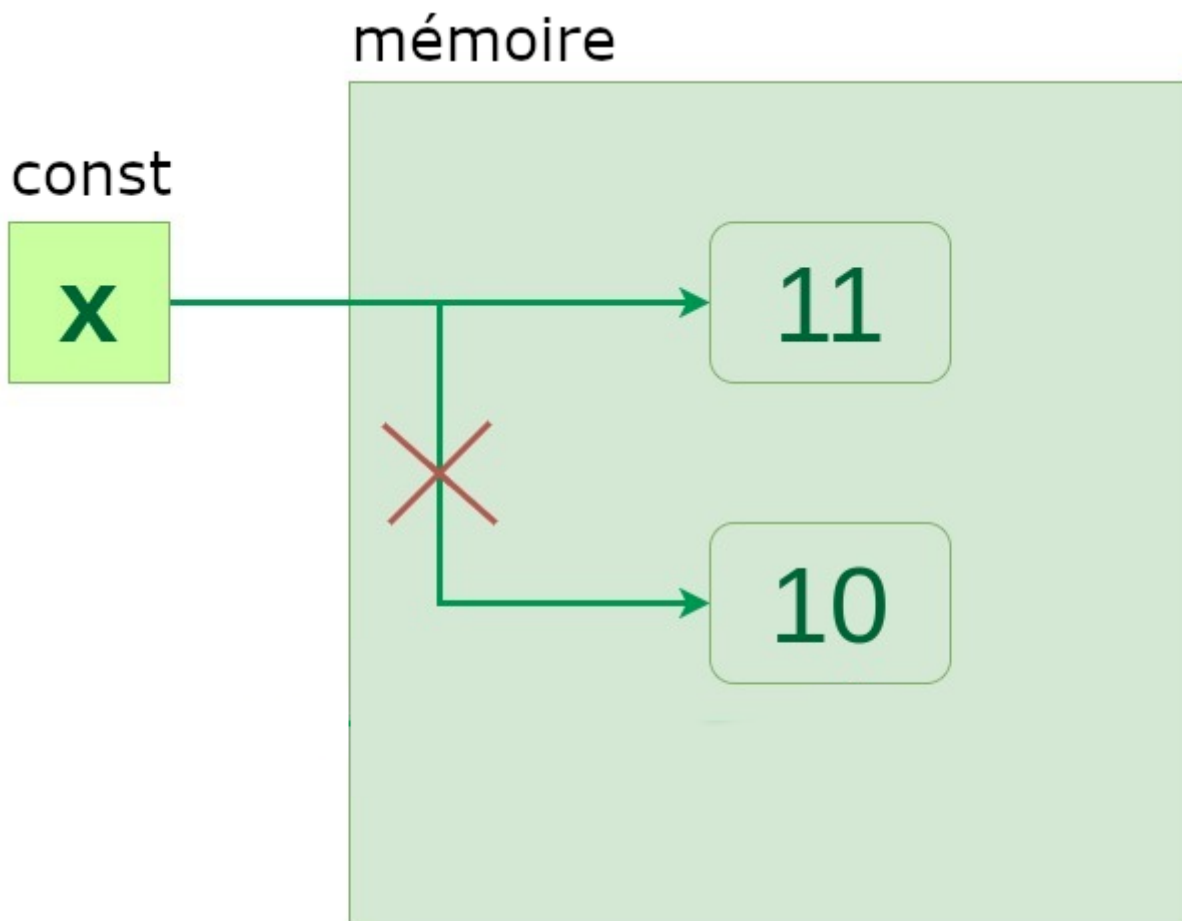
5.4.1.1 D'après vous que va afficher dans la console le code ci-dessus ?

5.5 Const

Le mot clé "const" sert à déclarer une constante. Cela veut simplement dire que vous ne pourrez pas réaffecter une nouvelle valeur à votre constante.

Ex :

```
const x = 11;  
x = 10;
```



✎ Exo

D'après vous que va afficher dans la console le code ci-dessus ?

6 Instructions, expressions et structures de contrôle

6.1 Instructions

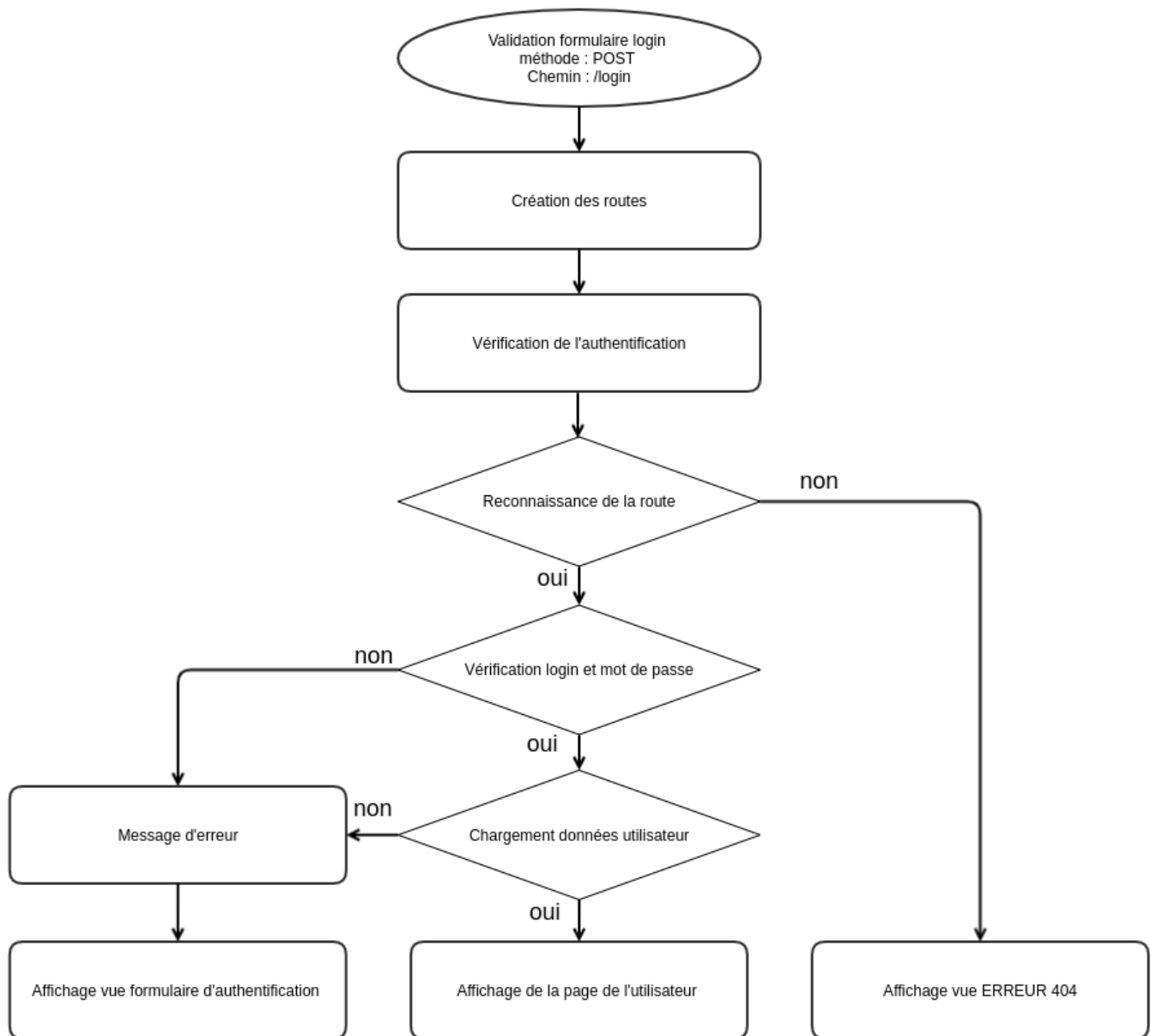
Les applications JavaScript sont composées de plusieurs instructions organisées grâce à une syntaxe. Une instruction

peut s'étaler sur plusieurs lignes et on peut avoir plusieurs instructions sur une seule ligne si chaque instruction est séparée de la suivante par un point-virgule.

6.1.1 Les instructions de contrôle du flux

Diagramme de flux

Lors de la création d'un programme, il est régulièrement nécessaire de faire des choix qui vont influencer le déroulement du parcours dans lequel est engagé l'utilisateur.



On peut visualiser ces parcours via des diagrammes de flux. Ces choix s'opèrent dans le code à l'aide de "structures" de contrôle ou instructions dont voici les principales

6.1.1.1 Condition `if ...else`

Cette instruction exécute une instruction si une condition donnée est vérifiée. Si la condition n'est pas vérifiée une autre instruction pourra être exécutée

```
let i = 2;
if(i > 2) {
    console.log("i est supérieur à 2");
}else {
    console.log("i est inférieur ou égal à 2");
}
/* == ceci est une comparaison */
if(i == 2){
    console.log("i est égal à 2");
}
```

6.1.2 Bloc

Une instruction de bloc est utilisée pour regrouper zéro ou plusieurs instructions. Un bloc est délimité par une paire d'accolades.

6.1.3 break

Cette instruction termine la boucle ou l'instruction `switch` ou l'instruction `label` en cours et continue l'exécution sur l'instruction suivant l'instruction terminée.

6.1.4 continue

Cette instruction termine l'exécution des instructions dans la boucle courante, ou la boucle avec une étiquette correspondante, et continue l'exécution de la boucle dans l'itération suivante.

6.1.5 switch

Cette instruction permet d'évaluer une expression et de faire correspondre le résultat de cette expression avec différents cas et d'exécuter les instructions associées aux cas qui ont chacun un identifiant.

L'instruction switch évalue une expression et, selon le résultat obtenu et le cas associé, exécute les instructions correspondantes.

```
const expr = 'Papayas';
switch (expr) {
case 'Oranges':
console.log('Oranges are $0.59 a pound. ');
break;
case 'Mangoes':
case 'Papayas':
console.log('Mangoes and papayas are $2.79 a pound. ');
// Expected output: "Mangoes and papayas are $2.79 a
pound."
break;
default:
console.log( Sorry, we are out of ${expr}. );
}
```

Que se passe-t-il si on oublie un break ?

Si on omet une instruction break, le script exécutera les

instructions pour le cas correspondant et aussi celles pour les cas suivants jusqu'à la fin de l'instruction switch ou jusqu'à une instruction break. Par exemple :

```
var toto = 0;
switch (toto) {
case -1:
console.log('moins un');
break;
case 0: // toto vaut 0 donc ce cas correspond
console.log(0);
// NOTE : le break aurait du être placé ici
case 1: // pas de break pour 'case 0:' les instructions de ce
cas sont
// exécutées aussi
console.log(1);
break; // on a un break a ce niveau donc les instructions
// des cas suivants ne seront pas exécutées
case 2:
console.log(2);
break;
default:
console.log('default');
}
```

6.1.6 throw

Cette instruction lève une exception.

6.1.7 `try...catch`

Cette instruction permet de spécifier un ensemble d'instructions à tenter, et de préciser le traitement à effectuer dans le cas où une exception est produite.

6.1.8 Déclarations

6.1.8.1.1 `var`

Cette instruction permet de déclarer une variable, éventuellement en fournissant une valeur pour permettant de l'initialiser.

6.1.8.1.2 `let`

Cette instruction permet de déclarer une variable locale dans une portée d'un bloc et éventuellement d'initialiser sa valeur.

6.1.8.1.3 `const`

Cette instruction déclare une constante en lecture seule.

6.2 Fonctions et classes

6.2.1.1.1 `function`

Cette instruction déclare une fonction avec les paramètres donnés.

6.2.1.1.2 `return`

Cette instruction spécifie la valeur de retour renvoyée par une fonction.

6.2.1.1.3 `class`

Déclare une classe.

6.2.1.2 Itérations

6.2.1.2.1 `for`

```
for(let i = 0; i < 10; i ++) {  
  console.log(i);  
}
```

6.2.1.2.2 `while`

Cette instruction permet de créer une boucle qui s'exécute tant qu'une condition de test est vérifiée. La condition est évaluée avant d'exécuter l'instruction contenue dans la boucle.

```
let i = 0;  
while(i < 10) {  
  console.log(2*i+1);  
  i++;  
}
```

6.2.1.2.3 `boucle do ... while`

Cette instruction crée une boucle qui s'exécute tant que la condition est vraie. La condition est évaluée après avoir exécuté une itération de boucle, ce qui fait que cette boucle sera exécutée au moins une fois.

6.2.1.2.4 `boucle for ... in`

```
const jc = {  
  nom: "Dusse",  
  prenom: "Jean-Claude",  
  sePresenter: function(){  
    console.log("Bonjour, je m'appelle " +  
      this.prenom + " " + this.nom);  
  }  
}  
  
for(let key in jc) {  
  console.log(key + " : " + jc[key]);  
}
```

// LES OBJETS EN JS SE COMPORTEMENT COMME DES
TABLEAUX ASSOCIATIFS (améliorés !)

```
console.log(jc.nom);  
console.log(jc["nom"]);  
console.log(jc.sePresenter());  
console.log(jcsePresenter);
```

6.2.1.2.5 Parcours d'un tableau à index avec la boucle for

```
// Tableau littéral à index
var personnages = ["Harry","Hermione","Ron","Voldemore"];

let taille = personnages.length;

for(var i = 0; i < personnages.length; i ++) {
  console.log(personnages[i]);
}
```

6.2.1.2.6 Parcours d'un tableau à index avec la boucle for ... of

```
const fruits = ["Cerise", "Pomme"];
for(let elt of fruits) {
  console.log(elt);
}
```

6.3 Autres

6.3.1.1.1 debugger

Cette instruction appelle une fonctionnalité de débogage. Si aucune fonctionnalité de débogage n'est disponible, l'instruction n'a aucun effet.

6.3.1.1.2 export

Cette instruction permet à un script signé de fournir des propriétés, fonctions et des objets à d'autres scripts (signés ou non).

6.3.1.1.3 `import`

Cette instruction permet à un script d'importer des propriétés, fonctions ou objets depuis un script qui les exporte.

6.3.1.1.4 `label`

Cette instruction fournit un identifiant auquel il est possible de se référer en utilisant une instruction `break` ou `continue`.

6.4 Expression

Une expression peut être vue comme une unité de code valide qui est résolue en une valeur. Il existe deux types d'expressions, celles qui ont des effets de bord (par exemple l'affectation d'une valeur) et celles qui sont purement évaluées.

L'expression `x = 7` fait partie de la première catégorie. Elle utilise l'opérateur `=` afin d'affecter la valeur sept à la variable `x`. L'expression elle-même est évaluée avec la valeur 7.

L'expression `3 + 4` fait partie de la deuxième catégorie. Elle utilise l'opérateur `+` afin d'ajouter 3 et 4 pour produire une valeur : 7. Toutefois, si cette valeur n'est pas utilisée au sein d'une structure plus importante (par exemple avec [une déclaration de variable](#) comme `const z = 3 + 4`), elle sera immédiatement écartée (il s'agit généralement dans ce cas

d'une erreur de programmation, car l'évaluation ne produira aucun effet).

Comme les exemples précédents ont permis de montrer, toutes les expressions complexes sont formées avec des *opérateurs*, tels que `=` ou `+`.

Une expression ne doit pas contenir de mot-clés comme `if`, `var`, etc

6.5 Expression VS Statement (instructions)

Les expressions produisent une valeur qui peut être transmise à une fonction. Les instructions ne produisent pas de valeur et ne peuvent donc pas être utilisées comme arguments de fonction.

6.6 Zoom sur quelques opérateurs remarquables

6.6.1.1.1 Chaînage optionnel (optional chaining) (`?.`)

[Voir la documentation](#)

L'opérateur `?.` fonctionne de manière similaire à l'opérateur de chaînage `.`, à ceci près qu'au lieu de causer une erreur si une référence est `null` ou `undefined`, l'expression se court-

circuite avec undefined pour valeur de retour. Quand il est utilisé avec des appels de fonctions, il retourne undefined si la fonction donnée n'existe pas.

Ex :

```
obj.last?.toUpperCase()
```

6.6.1.1.2 Opérateur OR (||)

```
let i = j || 12;
```

L'opérateur OR va retourner l'opérande de droite si celle de gauche est "falsy" (false, "", 0, undefined)

6.6.1.1.3 Comparaison

Cas des types primitifs

Dans le cas des types primitifs, js va comparer les valeurs (avec ou sans transtypage)

```
let i = 2;    // type : number
let j = "2";  // type : string
if(i==j){// le == fait du transtypage
    console.log("i est égal à j");
}
if(i===j){// le === ne fait pas de transtypage
    console.log("i est égal à j");
}else {
```

```
console.log("i n'est pas égal à j");  
}
```

6.6.1.2 A votre avis que va afficher le code ci dessus ?

Cas des types objet

Dans le cas des objets, js va comparer non pas les valeurs mais les références. [Plus d'infos](#)

```
let i = 2;  
let j = i;  
if (i === j) console.log('i et j identiques');  
  
let p1 = {"name": "Bob"};  
let p2 = {"name": "Bob"};  
let p3 = p1;  
if (p1 == p2) console.log('p1 et p2 identiques');  
if (p1 == p3) console.log('p1 et p3 identiques');
```

Exo

A votre avis que va afficher le code ci dessus ?

6.7 Voir le détail des autres opérateurs :

- Opérateurs d'affectation
- Opérateurs de comparaison
- Opérateurs arithmétiques
- Opérateurs binaires
- Opérateurs logiques
- Opérateurs pour les grands entiers
- Opérateurs pour les chaînes de caractères
- Opérateur conditionnel (ternaire)
- Opérateur virgule
- Opérateurs unaires
- Opérateurs relationnels

7 Fonctions

7.1 Fonction classique

une fonction permet d'isoler du code entre **accolades** (bloc de code) qui sera appelé via le nom de la fonction et avec des **arguments** dont la valeur va être assignée aux **paramètres**.

Une fonction doit être appelée pour être exécutée. Une fonction attend ou pas des **paramètres** en entrée et renvoie ou pas une valeur en sortie.

Un **paramètre** est une variable qui est déclarée lors de la définition de la fonction et dont la portée est le bloc de la

fonction.

Quand la fonction est appelée, les **arguments** sont les données que l'on assigne aux paramètres de la fonction.

La façon la plus classique , en js, de créer une fonction est d'utiliser le mot clé "function" suivie du nom de la fonction suivi de parenthèses dans les quelles on trouve les noms des paramètres séparés par des virgules puis enfin le corps de la fonction entouré d'accolades

Ex de fonction :

```
function logLastname(name) { // définition de la
fonction avec un paramètre
    console.log(name);
}
```

D'après vous que va afficher le code ci-dessus ?

7.2 Paramètres optionnels et valeurs par défaut

Si l'on passe plus d'arguments que de paramètres déclarés dans la fonction, le surplus est ignoré sauf à les récupérer via la variable **arguments**.

Si l'on passe moins d'arguments que de paramètres déclarés, les paramètres manquants auront pour

valeur **undefined**.

Depuis ES2015, on peut déclarer des paramètres optionnels de la façon suivante :

```
function getPerson(name, species = human) {  
  //  
}
```

7.3 Opérateur Rest

Depuis ES2015, il existe une nouvelle syntaxe pour accéder aux arguments

Ancienne méthode

```
const partners = [];  
function addPartner(people) {  
  for (let i = 0; i < arguments.length; i++) {  
    partners.push(arguments[i]);  
  }  
}  
addPartner("Bob", "Ray", "Nina");
```

Nouvelle méthode

```
function addPartner( ... people) {  
  people.forEach(person => {  
    partners.push(person);  
  });  
}
```

```
}  
}  
addPartner("Bob", "Ray", "Nina");
```

7.4 Hoisting

Le hoisting (en français, "hissage") est lié à la façon dont fonctionne les contextes d'exécution (précisément, les phases de création et d'exécution) en JavaScript. On peut résumer le mécanisme de hoisting en disant que les déclarations de variables et de fonctions sont déplacées physiquement en haut de votre code, même si ce n'est pas ce qui se passe en fait. A la place, les déclarations de variables et de fonctions sont mises en mémoire pendant la phase de compilation, mais restent exactement là où vous les avez tapées dans votre code.

L'un des avantages du fait que JavaScript met en mémoire les déclarations des fonctions avant d'exécuter un quelconque segment de code, est que cela vous permet d'utiliser une fonction avant que nous ne la déclariez dans votre code. Concrètement, c'est grâce au hoisting que le code suivant fonctionne :

```
afficheNomDeFamille("Gonzalez"); // appel de la  
fonction avec l'argument "Gonzalez"
```

```
function afficheNomDeFamille(nom){ // définition  
de la fonction  
    console.log(nom);  
}
```

Dans l'exemple ci-dessus, bien que la fonction soit déclarée après avoir été appelée, le code fonctionne !

7.5 Contexte d'exécution

Au départ du script, un contexte global d'exécution est créé. Il comprend les déclarations de fonction, leurs paramètres éventuels et les déclarations de variables utilisant var.

Ensuite le script s'exécute et affecte les valeurs aux différentes variables dans l'ordre du code. A chaque fois que le "js engine" rencontre une **fonction** (et aussi un **bloc de code** depuis ES6) ajoute un nouvel enregistrement dans la pile des contextes d'exécution.

contextes d'exécutions correspondants au code :

Contexte exécution {} de la fonction b() - k
Contexte exécution b() - l
Contexte exécution a() - j - b()
Contexte exécution global - i - a()



Exemple de code:

```
let i = 1;
function a() {
  let j = 2;
  b();
  function b(){
    {
      let k = 3;
    }
    let l = 4;
    console.log(l);
    console.log(k);
  }
}
```

```
}  
a();
```

7.5.1 Closure

La closure est l'expression de la capacité des fonctions à « capturer leur environnement »

Étudions l'exemple suivant :

```
function creerFonction() {  
  var nom = "Mozilla";  
  function afficheNom() {  
    console.log(nom);  
  }  
  return afficheNom;  
}  
  
let maFonction = creerFonction(); // à ne pas  
confondre avec let maFonction = creerFonction;  
maFonction();
```

"Mozilla" est affiché dans la console. L'intérêt de ce code est qu'une fermeture contenant la fonction `afficheNom` est renvoyée par la fonction parente, avant d'être exécutée.

Le code continue à fonctionner, ce qui peut paraître contre-intuitif au regard de la syntaxe utilisée. Usuellement, les variables locales d'une fonction n'existent que pendant

l'exécution d'une fonction. Une fois que `creerFonction()` a fini son exécution, on aurait pu penser que la variable `nom` n'est plus accessible. Cependant, le code fonctionne : en JavaScript, la variable est donc accessible d'une certaine façon.

L'explication est la suivante : `maFonction` est une fermeture (**closure**). La fermeture combine la fonction `afficheNom` et son environnement. Cet environnement est composé de toutes les variables locales accessibles (dans la portée) à la création de la fermeture. Ici `maFonction` est une fermeture qui contient la fonction `afficheNom` et une référence à la variable `var nom = "Mozilla"` qui existait lorsque la fermeture a été créée. L'instance de `afficheNom` conserve une référence à son environnement lexical, dans lequel `nom` existe. Pour cette raison, lorsque `maFonction` est invoquée, la variable `nom` reste disponible et "Mozilla" est transmis à `console.log`.

7.5.1.1 Autre exemple de closure

Extrait du [guide du bien développer en js de Ryan McDermott](#)

```
function makeBankAccount() {  
  // this one is private  
  let balance = 0;
```



```
// a "getter", made public via the returned
object below
function getBalance() {
    return balance;
}

// a "setter", made public via the returned
object below
function setBalance(amount) {
    // ... validate before updating the balance
    balance = amount;
}

return {
    // ...
    getBalance,
    setBalance
};
}
const account = makeBankAccount();
account.setBalance(100);
```

L'intérêt est clairement ici de rendre une variable privée mais tout de même accessible via un "getter" et un "setter"

7.6 Fonction anonyme immédiate

Les fonction anonymes immédiates ou Immediately invoked function expression ou IIFES permettent d'isoler le code et

donc les variables.

Comme nous l'avons vu dans le chapitre sur les variables, l'utilisation du mot clé "var" rend les variables "function scope". Cela veut dire que pour être sûr d'isoler son code, il faut l'encapsuler dans une fonction.

Une façon très répandue de faire et d'utiliser des fonctions anonymes immédiates dont voici la syntaxe :

```
(function() { //code isolé ici })();
```

C'est le fait de terminer l'instruction par les parenthèses ouvrantes et fermantes qui appelle immédiatement la fonction. Comme cette fonction n'a pas de nom, on dit qu'elle est anonyme.

7.6.1.1 Exercice

Examinez le code suivant :

```
(function(){  
    // code ici  
    console.log("Hello dans l'IIFES");  
    var i = 3;  
})();  
console.log(i);
```

D'après vous que va afficher la console ?

7.7 Arrow function

Une expression de fonction fléchée (arrow function en anglais) permet d'avoir une syntaxe plus courte que les expressions de fonction et n'a pas le même mécanisme d'affectation de "this". Ce dernier prendra la valeur du contexte de création de la fonction. Si la fonction fléchée est créée dans le contexte global, "this" sera alors "window" en revanche si elle est créée à l'intérieur d'une classe, "this" prendra la valeur de l'instance en cours de la dite classe.

Ex :

```
let a = () => {  
  console.log("Hello world");  
}
```

7.8 Raccourci de déclaration de méthode dans un objet

```
function createPerson() {  
  return {  
    walk() {  
      console.log('Je marche');  
    }  
  }  
}
```

```
}  
}  
}
```

7.8.1 First class citizen

Les fonctions sont des "first class citizen", elles peuvent être :

- stockées dans une variable,
- passées en argument à une fonction,
- retournées par une fonction.

7.8.2 Higher order function :

une "Higher order function" prend une fonction en paramètre ou renvoie une fonction ou les deux.

7.8.3 Fonction pure

Une fonction pure est une fonction qui possède les propriétés suivantes :

- Sa valeur de retour est la même pour les mêmes arguments (pas de variation avec des variables statiques locales, des variables non locales, des arguments mutables de type référence ou des flux d'entrée).

- Son évaluation n'a pas d'effets de bord (pas de mutation de variables statiques locales, de variables non locales, d'arguments mutables de type référence ou de flux d'entrée-sortie).

Une fonction pure est ainsi un analogue informatique d'une fonction mathématique.

Exemple de fonction pure :

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

Ces fonctions sont dites « pures » parce qu'elles ne tentent pas de modifier leurs entrées et retournent toujours le même résultat pour les mêmes entrées.

En revanche, cette fonction est impure car elle modifie sa propre entrée :

```
function add(account, amount) {  
  account.total += amount;  
}
```

7.8.4 setTimeout et setInterval

`setTimeout` et `setInterval` sont deux méthodes asynchrones qui permettent d'appeler des fonctions plus tard, un fois pour `setTimeout` et plusieurs fois à intervalles réguliers pour `setInterval`

La méthode `setInterval()`, proposée sur les interfaces `Window` et `Worker`, appelle de manière répétée une fonction ou exécute un extrait de code, avec un délai fixe entre chaque appel.

Paramètres : `setInterval` attend en paramètre une fonction et un délai en millisecondes.

Retour : Cette méthode renvoie un ID d'intervalle qui identifie de manière unique l'intervalle, vous pouvez donc le supprimer ultérieurement en appelant `clearInterval()`.

Exo

Ecrivez le code qui permet d'écrire dans la console 1, 2, 3, 4, 5 à une seconde d'intervalle en utilisant la fonction `setInterval`. Une fois arrivé à 5, la fonction qui affiche dans la console ne doit plus être appelée.

La méthode globale `setTimeout()` définit un temporisateur qui exécute une fonction ou un morceau de code spécifié une fois le temporisateur expiré.

Paramètres : setTimeout attend en paramètre une fonction et un délai en millisecondes.

Retour : Cette méthode renvoie l'id du timeout qui est une valeur entière positive qui identifie le temporisateur créé par l'appel à setTimeout(). Cette valeur peut être passée à clearTimeout() pour annuler le délai d'attente.

8 Objet

Si l'on en croit wikipedia, un objet est un conteneur symbolique et autonome qui contient des informations et des mécanismes concernant un sujet, manipulés dans un programme. Le sujet est souvent quelque chose de tangible appartenant au monde réel. C'est le concept central de la programmation orientée objet (POO).

En programmation orientée objet, un objet est créé à partir d'un modèle appelé **classe** ou **prototype**, dont il **hérite** les comportements et les caractéristiques. Les comportements et les caractéristiques sont typiquement basés sur celles propres aux choses qui ont inspiré l'objet : une personne (avec son état civil), un dossier, un produit...

Exemple

```
// Définir une fonction constructeur  
function **Personne**(nom,prenom) {
```

```
// Propriétés de l'objet
this.nom = nom;
this.prenom = prenom;
// Méthode (Camel case selon les standards)
this.sePresenter = function() {
    console.log("Bonjour, je m'appelle " +
        this.prenom + " " + this.nom);
}
}
//instanciation de l'objet et stockage dans la
variable bob
var bob = new Personne("Dylan", "Bob");
console.log(bob.nom);
console.log(bob.prenom);
bob.sePresenter();
// instanciation de la personne Jean-Claude Dusse
var jc = new Personne("Jean-Claude", "Dusse");
jc.sePresenter();
```

Vous noterez qu'il est classique de créer une instance d'objet puis d'appeler une méthode de l'objet via cette instance.

8.1 Qui est "this" ?

Dans le contexte global d'exécution (c'est-à-dire, celui en dehors de toute fonction), this fait référence à l'objet global **window**.

S'il est utilisé dans une fonction, la valeur de `this` dépendra de la façon dont la fonction a été déclarée et appelée.

Fonction déclarée avec le **mot clé function** (ex : `sePresenter: function(){} :`

- Si la fonction est appelée depuis une instance d'objet (comme c'était le cas dans l'exemple ci-dessus `jc.sePresenter()`), alors `this` prendra la valeur de l'instance en question.
- si la fonction n'est pas appelée depuis une instance d'objet, `this` redeviendra l'objet global **window**.

Fonction déclarée en utilisant la syntaxe des **arrow function** (ex : `sePresenter: () => {} :`

- `this` fait référence à l'instance de l'objet en cours si la fonction a été déclarée à l'intérieur d'une fonction constructeur (ou dans une classe comme on le verra plus tard)
- `this` fait référence à l'objet `window` si la fonction n'a pas été déclarée à l'intérieur d'une fonction constructeur (ou d'une classe comme on le verra plus tard)

8.2 Raccourci pour la création d'objets - depuis ES2015

Lorsque la propriété de l'objet que l'on veut créer a le même nom que la variable utilisée pour valeur, on peut utiliser un raccourci :

```
function createPerson() {  
  const name = 'bob';  
  return { name };  
}
```

8.3 Prototype

Le javascript est un langage à "prototype". Chaque objet possède une **propriété privée** qui contient un lien vers un autre objet appelé le **prototype**. Ce prototype possède également son prototype et ainsi de suite, jusqu'à ce qu'un objet ait null comme prototype. Par définition, null ne possède pas de prototype et est ainsi le dernier maillon de la chaîne de prototype.

8.4 Reprenons l'exemple de code du début de cette page :

```
function Personne(nom, prenom) {  
  // Propriétés de l'objet  
  this.nom = nom;  
  this.prenom = prenom;  
  // Méthode (Camel case selon les standards)
```

```
this.sePresenter = function() {  
    console.log("Bonjour, je m'appelle " +  
        this.prenom + " " + this.nom);  
}  
}
```

la méthode "sePresenter" sera créée pour chaque instance de "Personne" ce qui prend de la place inutilement en mémoire.

Pour corriger cela, il suffit d'ajouter la méthode au prototype de Personne de la manière suivante :

```
function Personne(nom, prenom) {  
    // Propriétés de l'objet  
    this.nom = nom;  
    this.prenom = prenom;  
}  
Personne.prototype.sePresenter = function() {  
    console.log("Bonjour, je m'appelle " +  
        this.prenom + " " + this.nom);  
}
```

Conclusion : le prototype permet de "factoriser" les propriétés d'un type d'objet. C'est d'ailleurs ce mécanisme qui est utilisé dans les "class" apportées par ES5. Ces dernières ne sont qu'un sucre syntaxique !

8.5 Exercice 1

Exo

Créer un constructeur de cercle qui a pour propriétés

Créer un constructeur de cercle qui a pour propriétés :

- "rayon" en mètre qui sera définie à l'instanciation de chaque cercle
- "nom" qui permettra de donner un nom à chaque cercle et qui sera définie à l'instanciation également de chaque cercle
- "Pi" qui sera stocké dans un seul espace mémoire (dans le prototype soit une propriété de classe)
- aire() qui affichera dans la console l'aire ($\pi \times \text{rayon}^2$).
- Créez 2 instances de Cercle, petit_cercle et grand_cercle qui auront respectivement pour rayon 2 et 4
- Appelez aire sur les 2 instances
- Essayer de définir au bon endroit "aire()"
- Puis instancier 2 cercles qui ont respectivement pour rayon : 2 et 4 mètres et pour nom petit_cercle et grand_cercle.:

En reprenant l'exemple du cercle, voici un dessin qui tente d'expliquer l'intérêt des prototypes.

Cas 1 : le prototype du constructeur cercle n'est pas utilisé. A chaque instance de cercle, on stocke la valeur pi et la méthode aire alors qu'elle sont les mêmes pour tous.

Constructeur cercle prototype

petit_cercle

rayon -> 10

nom -> petit cercle

pi -> 3.140

aire -> function() {//}

grand_cercle

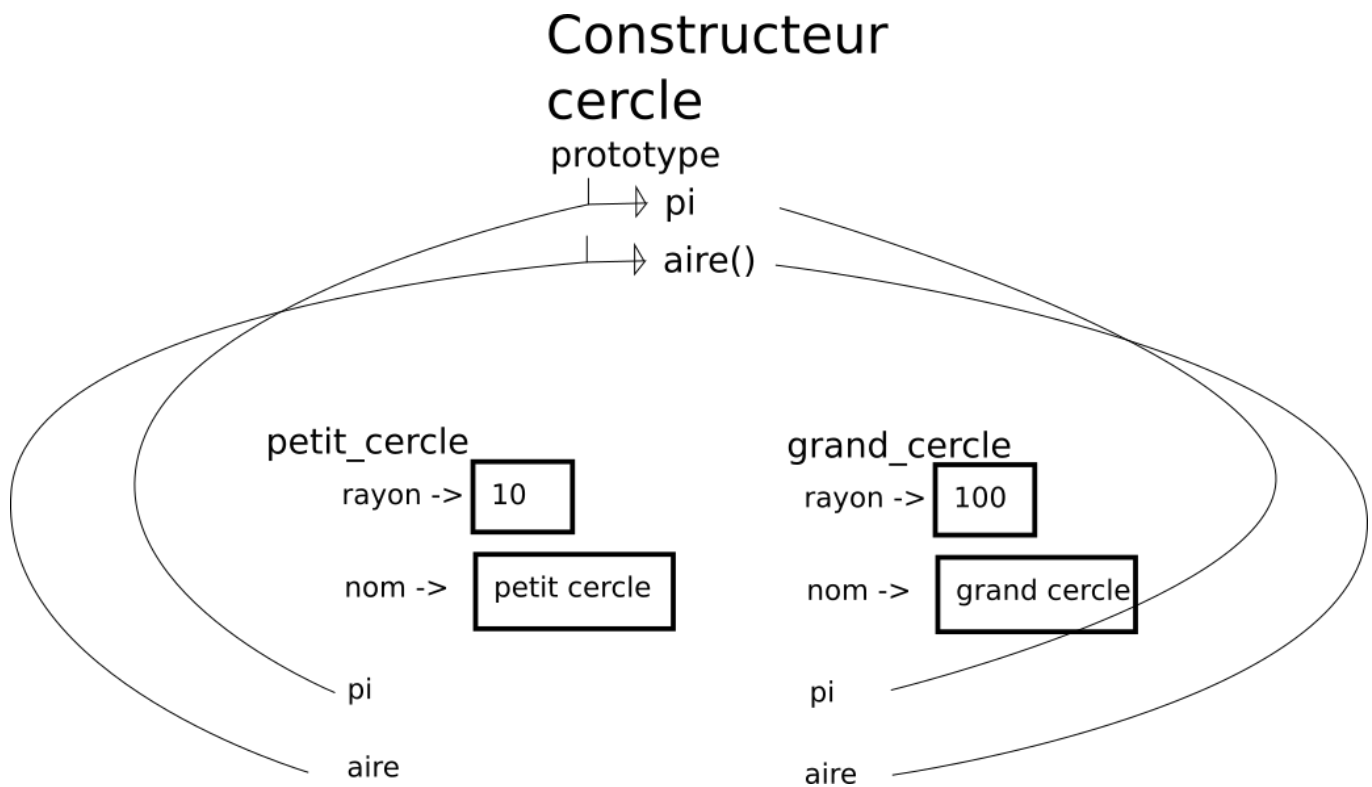
rayon -> 100

nom -> grand cercle

pi -> 3.140

aire -> function() {//}

Cas 2 : le prototype du constructeur cercle est utilisé. Pi et la méthode aire ne sont alors stockées qu'une seule fois.



8.5.1 Class et héritage

Depuis ECMAScript ES6, il est possible de créer des classes d'objets avec un mécanisme d'héritage

Ex :

```
// Création d'une "class" Personne ES6
class Personne { // Majuscule selon les standards
  constructor(nom, prenom) { // récupération des
    paramètres
    this.nom = nom; // propriété
    this.prenom = prenom; // propriété
  }
  // Méthodes ajoutées automatiquement au
  prototype de Personne
  sePresenter() {
```

```
        console.log("Bonjour, je m'appelle " +
            this.prenom + " " + this.nom);
    }
}

/**
 * instantiation d'une Personne avec passage
 * des paramètres "Chazal" et "Franck" au
 * constructeur
 */
var franck = new Personne("Chazal", "Franck"); //
franck.sePresenter();

// Création d'une "class" Enseignant qui hérite
// de la class Personne
class Enseignant extends Personne {
    constructor(nom, prenom, diplome) {
        super(nom, prenom);
        this.diplome = diplome;
    }
    // Méthodes
    sePresenter() {
        super.sePresenter();
        console.log(" ... et je suis un enseignant");
    }
    enseigner() {
        console.log("J'enseigne !");
    }
}

var jean = new
```

```
Enseignant("Dujardin", "Jean", "Agrégation");
jean.sePresenter();
jean.enseigner();

// Class qui spécialise la class Enseignant
class EnseignantProgrammation extends Enseignant {
    // Méthodes
    enseignerJS() {
        console.log("J'enseigne le JS !");
    }
}
var yvan = new
EnseignantProgrammation("Attal", "Ivan", "BAC");
yvan.sePresenter();
yvan.enseignerJS();
```

8.6 Propriétés privées avec getter et setter

Depuis ECMAScript 2020 (ES11), il est possible de gérer des propriétés privées avec getter et setter.

Références :

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields
- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get>

Exemple :

```
class Person {  
  #name;  
  constructor(name) {  
    this.#name = name;  
  }  
  get name() {  
    return this.#name;  
  }  
  set name(new_name) {  
    this.#name = new_name;  
  }  
}  
  
const b = new Person("Bob");  
console.log(b.name);  
b.name = "toto";  
console.log(b.name);
```

8.7 Propriétés et méthodes de classe avec le mot clé static

cf

: <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes/static>

Le mot-clé static permet de définir une propriété statique d'une classe. Les propriétés statiques ne sont pas

disponibles sur les instances d'une classe mais sont appelées sur la classe elle-même. Les méthodes statiques sont généralement des fonctions utilitaires (qui peuvent permettre de créer ou de cloner des objets par exemple).

8.8 Objet littéral

Il est possible (et fréquent) d'utiliser et de créer des objets de la manière suivante (syntaxe du JSON):

```
const jc = {  
  nom: "Dusse",  
  prenom: "Jean-Claude",  
  sePresenter: function(){  
    console.log("Bonjour, je m'appelle " +  
      this.prenom + " " + this.nom);  
  }  
}  
jc.sePresenter();
```

8.9 Assignment destructurée avec le spread operator

Depuis ES2015, on peut utiliser une nouvelle syntaxe, le spread operator, qui permet de copier et "d'étaler" des objets lors de l'assignation.

Ex :

```
const bob = {  
  firstname: "Bob"  
}  
const bobDylan = { ... bob, lastname: "Dylan" };  
console.log(`bobDylan`, bobDylan);
```

Attention il ne faut pas confondre le rest operator avec le spread operator même s'ils se ressemblent beaucoup.

8.10 Récupérer des informations sur les objets

- Tester la propriété d'un objet avec [hasOwnProperty](#).
- Copier un objet avec [assign](#)
- Copier un objet avec le [spread operator](#)
- Connaître le nom de la classe d'un objet

Vous vous rendrez compte qu'il n'est pas toujours évident de savoir exactement de quelle classe (ou fonction constructeur) est issue une instance. Voici un code qui répond à cette question :

```
console.log(mon_instance.constructor.name);
```

8.11 Connaître les noms des classes dont héritent une instance d'objet

Pour aller plus loin dans la connaissance d'un objet, ce code peut vous être utile

```
class A {}
class B extends A {}
class C extends B {}
function logClasses(object) {
  while (object) {
    object = Object.getPrototypeOf(object);
    if(object) console.log("classe :
",object.constructor.name);
  }
}
logClasses(new C());
```

Que va renvoyer le code ci dessus ?

```
class A {}
class B extends A {}
class C extends B {}
function logClasses(object) {
  while (object) {
    object = Object.getPrototypeOf(object);
    if(object) console.log("classe :
",object.constructor.name);
  }
}
```

```
}  
logClasses(new C());
```

8.12 Exercice 2

Exo

Ecrire une classe qui permet de créer des "Bike" qui auront pour propriétés "brand", "model", "weight" et pour méthode "pedal" qui renvoie simplement dans la console : "Je pédale !"

Réécrire la méthode "pedal" qui renvoie dans la console (uniquement pour les tandems) : "Nous sommes 2 à pédaler !"

Pour les plus avancés, faites en sorte que brand et model soient des propriétés privées et qui seront manipulées via un getter et un setter.

8.13 Exercice 3

Exo

Créer une classe "CustomString" qui étend la classe String. Ré-écrire la méthode "split" afin qu'elle renvoie deux tableaux :- Le premier contient exactement la même chose que :

- le retour de split telle que définie par défaut
- le deuxième contenant le tableau qui comprend le caractère qui a servi à couper la chaîne.
Ex split(o) de "Hello World" doit renvoyer :
- ['Hell', ' W', 'rld']
- ['Hello', ' Wo', 'rld']

9 Tableaux

Les tableaux sont des objets de haut-niveau semblables à des listes.

9.1 Créer un tableau et obtenir sa taille

```
const fruits = ['Apple', 'Banana'];  
console.log(fruits.length); // 2
```

9.2 Accéder (via son index) à un élément du tableau

```
const first = fruits[0]; // Apple  
const last = fruits[fruits.length - 1]; // Banana
```

9.3 Boucler sur un tableau

```
fruits.forEach(function(item, index, array) {  
  console.log(item, index); // Apple 0 puis // Banana 1  
});
```

9.4 Ajouter à la fin du tableau

```
const newLength = fruits.push('Orange');  
// ["Apple", "Banana", "Orange"]
```

9.5 Supprimer le dernier élément du tableau

```
const last = fruits.pop(); // supprime Orange (à la fin)  
// ["Apple", "Banana"];
```

9.6 Supprimer le premier élément du tableau

```
const first = fruits.shift(); // supprime Apple (au début)  
// ["Banana"];
```

9.7 Ajouter au début du tableau

```
const newLength = fruits.unshift('Strawberry') // ajoute au  
début  
// ["Strawberry", "Banana"];
```

##. Trouver l'index d'un élément dans le tableau

```
fruits.push('Mango');  
// ["Strawberry", "Banana", "Mango"]  
const pos = fruits.indexOf('Banana');  
// 1
```

9.8 Trouver l'index d'un élément dans le tableau en fonction d'une condition

[Grâce à la méthode findIndex](#)

Ex :

```
let term_index = state.terms.findIndex(element => {  
  return element.id === termId;  
});
```

9.9 Méthode "map"

9.9.1.1 où comment créer un nouveau tableau à partir d'un tableau existant selon une fonction de transformation

La méthode map() crée un nouveau tableau avec les résultats de l'appel d'une fonction fournie sur chaque élément du tableau appelant.

Ex :


```
const array1 = [1, 4, 9, 16];  
// pass a function to map  
const map1 = array1.map(x => x * 2);  
  
console.log(map1);  
// expected output: Array [2, 8, 18, 32]
```

9.9.2 Méthode "filter"

9.9.2.1 où comment créer un nouveau tableau à partir d'un tableau existant en filtrant selon une condition

Ex :

```
const words = ['spray', 'limit', 'elite',  
               'exuberant', 'destruction', 'present'];  
const result = words.filter(word => word.length >  
6);  
console.log(result);  
// expected output: Array ["exuberant",  
"destruction", "present"]
```

##Supprimer un élément par son index

```
const removedItem = fruits.splice(pos, 1); //  
supprime 1 élément à la position pos
```

```
// ["Strawberry", "Mango"]
```

9.9.3 Supprimer des éléments à partir d'un index

```
const vegetables = ['Cabbage', 'Turnip', 'Radish', 'Carrot'];
console.log(vegetables);
// ["Cabbage", "Turnip", "Radish", "Carrot"]
var pos = 1, n = 2;
var removedItems = vegetables.splice(pos, n);
// n définit le nombre d'éléments à supprimer,
// à partir de la position pos
console.log(vegetables);
// ["Cabbage", "Carrot"] (le tableau d'origine est changé)

console.log(removedItems);
// ["Turnip", "Radish"] (splice retourne la liste des éléments
supprimés)
```

9.10 Copier un tableau

```
const shallowCopy = fruits.slice(); // crée un
nouveau tableau qui contient les éléments de
fruits
// ["Strawberry", "Mango"]
```

9.11 Trier un tableau

Pour trier un tableau, on va utiliser une fonction de callback qui attend deux arguments. Les deux paramètres vont permettre de comparer les éléments deux par deux. Le classement des éléments du tableau est basé sur la valeur de retour de la fonction de callback.

Si la fonction retourne une valeur :

- **> 0** alors, l'ordre des éléments sera inversé
- **< 0**, l'ordre des éléments restera inchangé

Attention, contrairement à filter ou à map, sort modifie le tableau initial

9.11.1.1 Exemple

```
const tableau = [{"id":2}, {"id":5}, {"id":1}];

tableau.sort(function (a, b) {
  if (a.id < b.id)
    return -1;
  if (a.id > b.id)
    return 1;
  // a doit être égal à b
  return 0;
});
console.log("tableau trié par ordre d'id croissant
: ", tableau);

// version plus courte de la fonction de
```

comparaison :

```
tableau.sort(function (a, b) {  
    return a.id - b.id;  
});
```

[+ d'infos](#)

9.11.2 Assignment destructurée avec le rest operator

Depuis ES2015, on peut utiliser une nouvelle syntaxe, le rest operator (...), soit pour récupérer des arguments d'une fonction, soit pour les assignments destructurées :

```
const fruits = ["Banane", "Cerise", "Pomme"];  
const [banane, ... otherFruits] = fruits;  
console.log(`banane`, banane);  
console.log(`otherFruits`, otherFruits);
```

Attention il ne faut pas confondre le rest operator avec le spread operator même s'ils se ressemblent beaucoup.

9.11.3 Assignment avec le spread operator

Comme son nom l'indique le spread operator prend un tableau et l'étale. Ex :

```
const exoticFruits = ["Banane", "Mangue",  
"Litchi"];  
const fruits = [ ... exoticFruits, "Cerise",  
"Pomme", "Abricots"];
```

9.12 Exercice 1 : Classer des utilisateurs par age

Classer (sort) le tableau suivant par age croissant puis par age décroissant.

```
const users = [  
  {name: "Dylan", age: 78},  
  {name: "Marley", age: 92},  
  {name: "Cohen", age: 83},  
  {name: "Jackson", age: 76},  
]
```

9.13 Exercice 2 - Chaînage des méthodes

Exo

En une seule instruction, ajouter 3 fruits au tableau "fruits" et classer les fruits par taille de chaîne de caractères.

N'utilisez pas push mais plutôt le spread operator et ensuite sort et afficher le résultat dans la console.

9.14 Exercice 3 - Méthode reduce

Exo

Lire la documentation sur la méthode "reduce" et nous expliquer le fonctionnement, exemple à l'appui.

10 Asynchrone

En javascript, la notion d'asynchronisme est omniprésente. Que ce soit pour la gestion des événements ou la récupération de données ou simplement avec des méthodes comme setTimeout, il est pratiquement impossible d'y échapper !

Premier exemple

```
console.log('avant');  
setTimeout(() => {console.log('affiché après 2  
secondes');},2000);  
console.log('après');
```

Vous constatez que le message "affiché après 2 secondes" s'affiche en dernier. On peut en conclure que la méthode

setTimeout est "non bloquante", c'est à dire qu'il n'est pas nécessaire d'attendre son résultat pour passer à l'exécution de la ligne suivante. On dit alors qu'elle est **asynchrone**.

10.1 "Fil d'exécution" ou thread

Cela va nous permettre d'introduire la notion de "fil d'exécution" ou thread. Pour comprendre cette notion, on fait souvent la comparaison avec un serveur dans un restaurant : lorsque vous commandez, le serveur part en cuisine, passe la commande et retourne en salle pour prendre d'autres commandes. On comprend que la commande est "non bloquante" ou asynchrone, c'est à dire que le serveur n'a pas besoin d'attendre que le plat soit prêt pour continuer son travail. Quand votre plat sera prêt, c'est toujours le même serveur qui vous servira votre plat. On dit alors que le service est mono-thread en opposition à muti-thread.

Il en est de même pour le javascript classique qui est **mono-thread**, ce qui signifie qu'il n'y a qu'une **seule pile d'exécution**.

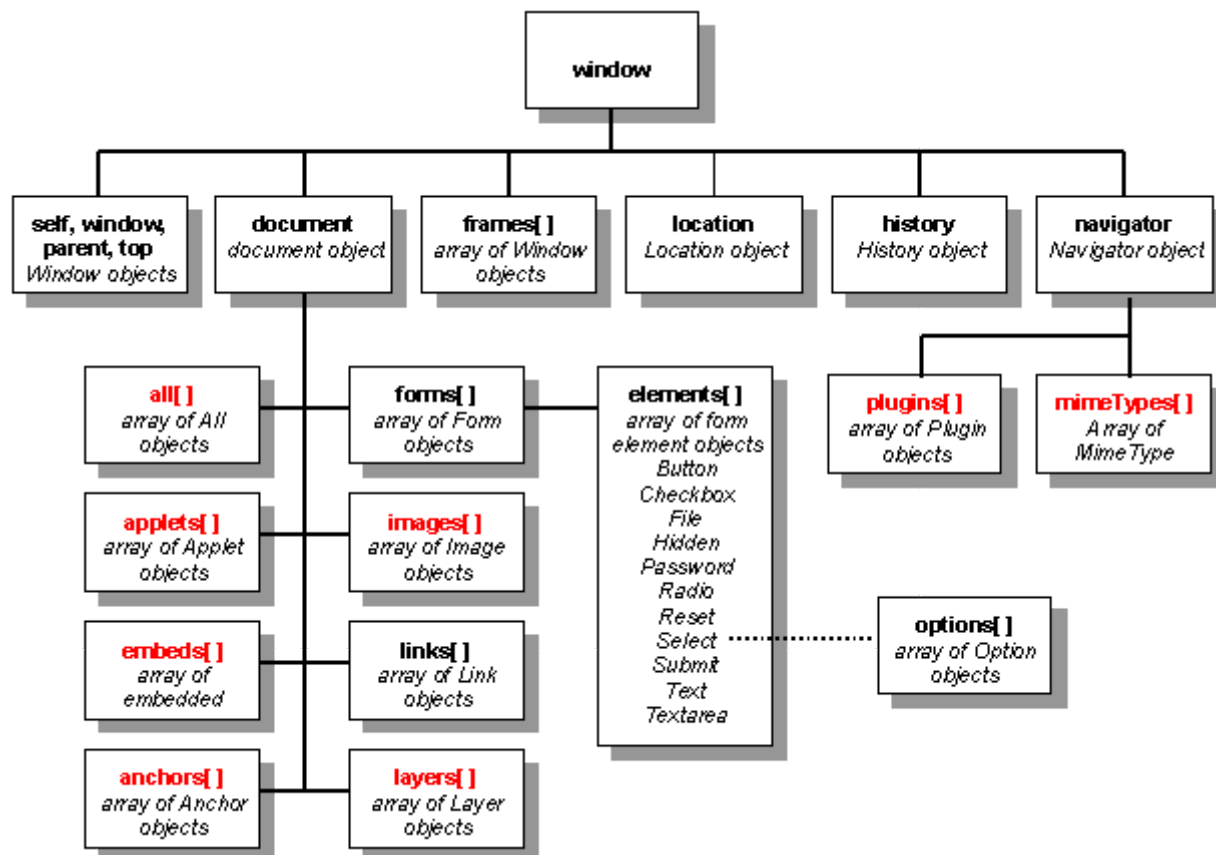
11 Document Object Model (DOM)

11.1 Références :

- Document
: <https://developer.mozilla.org/fr/docs/Web/API/Document>
- Node
: <https://developer.mozilla.org/fr/docs/Web/API/Node#m%C3%A9thodes>
- Element (élément du dom)
: <https://developer.mozilla.org/fr/docs/Web/API/Element>

Le Document Object Model (DOM) est une interface de programmation pour les documents HTML. Il fournit une page dont des programmes peuvent modifier la structure, son style et son contenu. Cette représentation du document permet de le voir comme un groupe structuré de nœuds et d'objets possédant différentes propriétés et méthodes. Fondamentalement, il relie les pages Web aux scripts ou langages de programmation.

11.2 Représentation générique du DOM

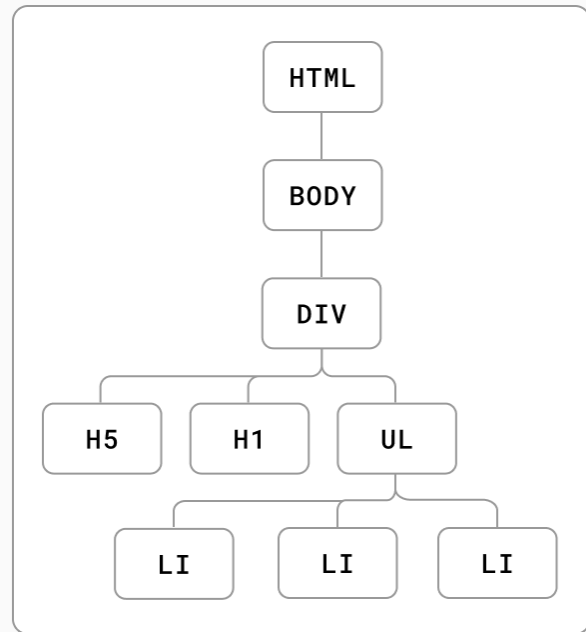


11.3 Un exemple de construction du DOM par un navigateur à partir d'un fichier HTML

HTML

```
index.html
1 <html>
2   <body>
3     <div>
4       <h5>Meet the</h5>
5       <h1>Engineers</h1>
6       <ul>
7         <li>A. Lovelace</li>
8         <li>G. Hopper</li>
9         <li>M. Hamilton</li>
10      </ul>
11    </div>
12  </body>
14 </html>
15
16
```

DOM



Il existe de nombreuses méthodes js pour accéder et modifier le DOM. Vous trouverez ci-dessous une sélection des propriétés indispensables à connaître par coeur :

Accéder à un élément en utilisant son identité

```
const element = document.getElementById("id-de-l-element-html");
```

Supprimer un élément du DOM

```
element.remove();
```

Créer un élément du DOM (ici une section)

```
const section = document.createElement("section");
```

Ajouter un élément à l'arbre du document (DOM), ici le body

```
window.document.body.appendChild(section);
```

Ajouter du texte à un élément du dom (section par exemple)

```
section.innerText = "Texte à ajouter";
```

Ajouter un attribut (ici l'identité "news" à l'élément stocké dans la const section)

```
section.setAttribute("id", "news");
```

Récupérer la valeur d'un attribut

```
section.getAttribute("id");
```

11.4 Héritage des éléments du DOM

Imaginons que votre page html contienne un balise h2 ayant pour valeur de l'attribut id "h2". Pour récupérer une référence à cet élément du dom, vous pouvez vous y prendre comme ceci :

```
let h2 = document.getElementById("h2");
```

Si vous êtes curieux, vous pouvez chercher à savoir de quel classe est issu cet élément :

```
console.log("class de h2 : ",  
h2.constructor.name); // affiche class de h2 :  
HTMLHeadingElement
```

Si vous êtes très curieux, vous pouvez remonter la chaîne des prototypes pour savoir exactement de qui h2 hérite :

```
while (h2) {  
    console.log("class de h2 : ",  
h2.constructor.name);  
    // Remonte la chaîne des prototypes  
    h2 = Object.getPrototypeOf(h2);  
}
```

[En vérifiant dans la documentation de Mozilla](#), vous pouvez en apprendre plus sur chacune des interfaces dont héritent les éléments du DOM :

L'interface `HTMLHeadingElement` représente les différents éléments d'en-tête `<h1>` à `<h6>`. Elle hérite des méthodes et des propriétés de l'interface [HTMLInputElement](#).

EventTargetNodeElementHTMLElementHTMLHeadingElement

11.5 Sélection avancée des éléments du DOM

[document.querySelector\(".open-close > h2"\);](#) permet de récupérer **le premier élément** du DOM qui correspond au sélecteur passé en paramètre

[document.querySelectorAll\(".open-close > h2"\);](#) permet de **récupérer un tableau d'éléments** qui correspondent au sélecteur passé en paramètre

[elt.nextElementSibling](#) : permet de récupérer l'élément suivant l'élément elt

11.6 Exercice 1

Retrouvez à quelle interface du DOM appartiennent les propriétés suivantes :

- innerText
- focus
- insertBefore
- innerHTML
- classList
- addEventListener

Profitez-en pour les retenir, elles vous serviront dans votre vie de développeur front-end !

11.7 Exercice 2

Créer en js une balise "nav" qui contient 4 boutons avec les textes "Item 1", "Item 2", "Item 3", Item 4". Placez cet élément du dom dans le header.

Faites en sorte que le premier item soit de couleur rouge (utilisez `querySelector` puis la propriété `"style.color"`).

Utilisez pour cela uniquement du javascript. Ré-utilisez la fonction `createMarkup` dont voici le code :

```
/**
 * Crée un élément du dom, lui ajoute du texte,
 * le place comme dernier
 * enfant de parent et ajoute un attribut en
 * utilisant le paramètre attributes
 * @param {String} markup_name
 * @param {String} text
 * @param {domElement} parent
 * @param {Object} attributes
 * @returns domElement
 */
function createMarkup(markupname, text, parent,
attributes = {}) {
  const markup =
document.createElement(markupname);
```

```
markup.textContent = text;
parent.appendChild(markup);
for (key in attributes) {
    markup.setAttribute(key, attributes[key]);
}
return markup;
}
```

11.8 Exercice 3

Améliorez la fonction createMarkup de façon à ce que l'on puisse ajouter plusieurs attributs. Le paramètre attribute sera remplacé par "attributes" et ne sera plus de type "Objet" mais de type "Array".

Pour finir, faites en sorte que cette fonction n'attende qu'un seul paramètre de type "Object"

11.9 Exercice 4

Le but de cet exercice est de créer un simple diaporama qui permet de faire défiler des images.

Voici comment agencer votre code :

- Créer une classe Slideshow
- le constructeur de cette classe attend 4 paramètres :
 - nb_images (le nombre d'images que va gérer le slideshow,

- width (la largeur du slideshow),
 - height (la hauteur du slideshow),
 - speed (la vitesse de changement d'images en millisecondes).
- Dans le constructeur, vous initialisez 5 propriétés :
 - nb_images (number)
 - images (array)
 - width (number)
 - height (number)
 - speed (number)
 - puis, toujours dans le constructeur, vous faites appel à trois méthodes
 - feedSs (// remplissage du tableau d'images "images")
- Pour créer des images (élément du DOM "img"), vous utiliserez la méthode suivante :

```
createImage = function() {  
    // création d'une image  
    const img =  
document.createElement("img");  
  
img.setAttribute("src", `https://picsum.ph  
otos/${this.width}/${this.height}?  
id=${Math.random()*1000}`);
```



```
        return img;
    }
```

- render (rendu du slideshow)
- animateSs(animation du slideshow)
- Pour l'animation, vous serez amené à utiliser soit `setTimeout` soit `setInterval`

Try catch

L'instruction `try ... catch` regroupe des instructions à exécuter et définit une réponse si l'une de ces instructions provoque une exception.

```
```js
```

```
try {
 nonexistentFunction();
} catch (error) {
 console.error(error.message);
}
```

Les exceptions peuvent être levées volontairement (`throw`) ou involontairement par l'interpréteur js.

Il est également possible de créer sa propre erreur :

```
throw new Error("oops");
```

L'instruction `throw` permet de lever une exception définie par l'utilisateur. L'exécution de la fonction courante sera stoppée (les instructions situées après l'instruction `throw` ne seront pas exécutées) et le contrôle sera passé au premier bloc `catch` de la pile d'appels. Si aucun bloc `catch` ne se trouve dans les fonctions de la pile d'appels, le programme sera terminé.

## 11.9.1 Fonction qui renvoie null en cas d'erreur

Il peut parfois être intéressant de faire en sorte qu'une erreur ne stoppe pas l'exécution d'un script mais que la fonction incriminée renvoie `null`. Ex :

```
function getI() {
 try {
 return i;
 } catch (error) {
 return null;
 }
}
console.log(`retour de getI`, getI());
```

# 12 Événements

Les événements permettent de déclencher une fonction après un ... événement ! Il peut s'agir classiquement d'un clic sur un bouton.

## 12.1 Liste des événements

Voici la liste des événements principaux, ainsi que les actions à effectuer pour qu'ils se déclenchent :

Nom de l'événement	Action pour le déclencher
<code>click</code>	Cliquer (appuyer puis relâcher) sur l'élément
<code>dblclick</code>	Double-cliquer sur l'élément
<code>mouseover</code>	Faire entrer le curseur sur l'élément
<code>mouseout</code>	Faire sortir le curseur de l'élément
<code>mousedown</code>	Appuyer (sans relâcher) sur le bouton gauche de la souris sur l'élément
<code>mouseup</code>	Relâcher le bouton gauche de la souris sur l'élément
<code>mousemove</code>	Faire déplacer le curseur sur l'élément
<code>keydown</code>	Appuyer (sans relâcher) sur une touche de clavier sur l'élément
<code>keyup</code>	Relâcher une touche de clavier sur l'élément
<code>keypress</code>	Frapper (appuyer puis relâcher) une touche de clavier sur l'élément

Nom de l'événement	Action pour le déclencher
<code>focus</code>	« Cibler » l'élément
<code>blur</code>	Annuler le « ciblage » de l'élément
<code>change</code>	Changer la valeur d'un élément spécifique aux formulaires ( <code>input</code> , <code>checkbox</code> , etc.)
<code>input</code>	Taper un caractère dans un champ de texte
<code>select</code>	Sélectionner le contenu d'un champ de texte ( <code>input</code> , <code>textarea</code> , etc.)

## 12.2 Exemple

// gestion de l'événement click sur div1

// Récupération d'un élément du DOM

let h1 = window.document.getElementById("h1");

/\*\*

*Gestion de l'événement click sur h1*

On assigne à la propriété "onclick"

*une méthode appelée lors d'un click sur l'objet en question.*

*Le "this" devient alors l'objet en question*

/

h1.onclick = function() {

console.log("click sur le h1");

```
console.log(this);
};
```

## 12.3 Objet événement

l'objet événement (qui correspond à l'[interface Event](#)) est automatiquement transmis aux gestionnaires d'événements pour fournir des fonctionnalités et des informations supplémentaires. Une interface définit les méthodes à implémenter, elle permet de définir un contrat : chaque classe implémentant l'interface sera tenue d'implémenter les méthodes de l'interface. [Voir les méthodes et propriétés de l'interface Event](#).

Plusieurs paramètres

### 12.3.1 Exemple de récupération de l'objet événement

```
// gestion de l'événement click sur h1
h1.onclick = function(e) {
 console.log("click sur le h1");
 console.log(e.target);
};
```

### 12.3.2 Exemple de passage de 2 paramètres à la fonction qui est

# déclenchée au click sur h1

```
h1.onclick = function(e) {
 manageClick(e, "Hello");
};
const manageClick = function(e, j){
 console.log("click sur le h1");
 console.log(e.target);
 console.log(j);
}
```

## 12.4 addEventListener

### [Documentation](#)

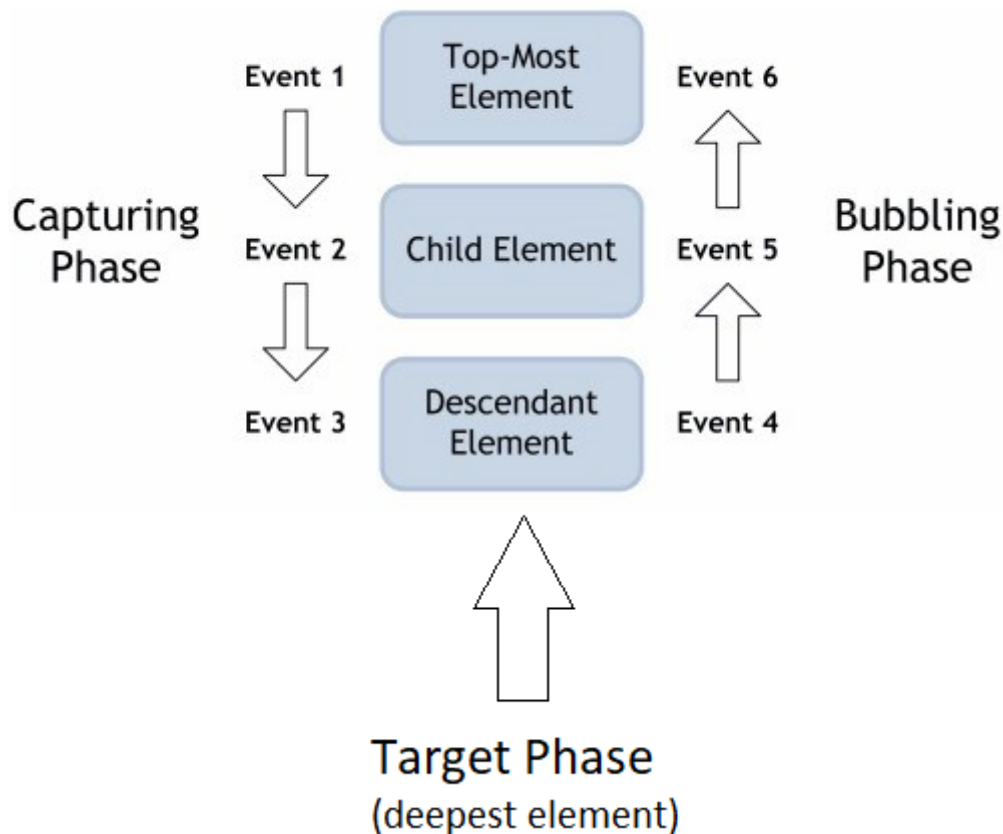
La méthode `addEventListener` présente deux avantages principaux par rapport à l'utilisation de propriétés de type `"onclick"` :

1. elle permet à un élément du dom d'écouter plusieurs événements de même type
2. elle attend un troisième paramètre qui permet de gérer plus finement les événements

La méthode `addEventListener` est appelée depuis une cible (un élément du dom en général) et attend trois paramètres :

1. le type d'événement (click, hover, ...)

2. une fonction à appeler chaque fois que l'événement spécifié dans le premier argument est envoyé à la cible
3. historiquement, le troisième paramètre de `addEventListener` était un boolean qui indiquait s'il fallait ou non utiliser la "capture". Cette dernière peut être définie comme la phase descendante de la propagation de l'événement en opposition à la phase montante ou "bubbling phase" .



Plutôt que d'ajouter davantage de paramètres à la fonction, le troisième paramètre a été changé en un objet pouvant contenir diverses propriétés définissant les valeurs des options pour configurer le processus de suppression de l'écouteur d'événement.

Exemple de code :

## HTML

```
<div id="outer">
 <div id="inner">
 <div id="target">Click ici</div>
 </div>
</div>
```

```
const outer = document.getElementById("outer");
const inner = document.getElementById("inner");
const target = document.getElementById("target");
```

```
outer.addEventListener("click", (e) =>
{console.log("outer en descendant",
e.eventPhase)}}, true);
outer.addEventListener("click", (e) =>
{console.log("outer en montant", e.eventPhase)}},
false);
```

```
inner.addEventListener("click", (e) =>
{console.log("inner en descendant",
e.eventPhase)}}, true);
inner.addEventListener("click", (e) =>
{console.log("inner en montant", e.eventPhase)}},
false);
```

```
target.addEventListener("click", (e) =>
```



```
{console.log("inner en descendant",
e.eventPhase);}, true);
target.addEventListener("click", (e) =>
{console.log("inner en montant", e.eventPhase);},
false);
```

## 12.5 Gestion des événements clavier

référence

: [https://developer.mozilla.org/fr/docs/Web/API/Element/keydown\\_event](https://developer.mozilla.org/fr/docs/Web/API/Element/keydown_event)

L'événement `keydown` est déclenché lorsque l'utilisatrice ou l'utilisateur appuie sur une touche du clavier.

### 12.5.1 Exemple

Imaginons que nous ayons une liste d'images cliquables grâce à la fonction `addEventListener`. La bonne pratique pour rendre cette fonctionnalité accessible sera de :

1. rendre ces images "focusables" via la touche tab
2. donner une alternative à l'événement "click" en utilisant l'événement "keydown" et en vérifiant que c'est bien la touche "Enter" qui a été la dernière utilisée.

## 12.6 Exercices

### 12.6.1 Paragraphes Lorem ipsum

En js, ajouter un bouton et une section dans le body. Le bouton aura pour intitulé : "Ajouter un paragraphe". Au click sur ce bouton, un nouveau paragraphe sera ajouté comme dernier enfant de la section qui comprendra tous les paragraphes. Chaque paragraphe aura "Lorem ipsum ..." comme texte.

## 12.6.2 Liste de tâches

En js, ajouter un formulaire dans le body comprenant une zone de texte (input) et un bouton "Ajouter une tâche". Au click sur ce bouton, et après avoir renseigné l'input du formulaire (  ), une tâche (ex : acheter du pain ) apparaîtra juste sous le formulaire et dans une section.

Chaque tâche est créée avec la balise section et permettra d'afficher côte à côte :

- l'intitulé de la tâche,
- un bouton pour valider la tâche (cette dernière sera alors barrée, un bouton "invalider" remplace le bouton "valider" et toute la tâche est déplacée en fin de section qui entoure l'ensemble des tâches
- un bouton pour supprimer la tâche qui déclenchera au click une "pop-up" grâce à la méthode "confirm()" pour que l'internaute confirme son choix de suppression. En

cas de confirmation, la section "tâche" correspondante sera supprimée.

Ajouter une tâche

Faire du sport

validerSupprimer

Planter des salades

validerSupprimer

~~Offrir des fleurs à Simone~~

invaliderSupprimer

## 13 Modules

Depuis ES6, on peut gérer les dépendances entre fichiers avec les mots clés "import" et "export"

Ex

```
export default class Person {
 constructor(name) {
 this.name = name;
 }
 present() {
 console.log("hello, I'm " + this.name);
 }
}
```

De cette façon, dans un autre script, on pourra avoir :

```
import Person from "./Person.js";
const p = new Person("Bob")
```

Attention, il faudra penser à appeler votre js en utilisant l'attribut type="module"

```
<script type="module" src="test.10-module.js">
```

## 13.1 Exports et imports multiples

Prenons l'exemple d'un fichier consts.js qui définit les deux constantes pi et nb\_or :

```
export const pi = 3.14159265359;
export const nb_or = 1.61803398875;
```

Il existe deux syntaxes pour importer ces constantes :

```
import { pi, nb_or } from "./consts.js";
console.log("pi : ", pi);
console.log("nb_or : ", nb_or);
```

ou

```
import * as consts from "./consts.js";
console.log("pi : ", consts.pi);
console.log("nb_or : ", consts.nb_or);
```

## Propriétés privées

Imaginons que nous avons besoin d'une fonction

"createStore" qui permet de créer une propriété state.

Imaginons que cette propriété ne doivent pas être accessible en modification directement depuis un autre endroit que la fonction "createStore" elle même.

Voici comment on pourrait s'y prendre :

Fichier store.js

```
function createStore() {
 let state = 0;
 function getState() {
 return state;
 }
 return {
 getState
 }
}
export default createStore();
```

Fichier main.js

```
import store from "./store.js";

console.log("store.state : ", store.getState());
```

Vous constaterez qu'**il n'**est pas possible de modifier directement la propriété `state` de `store` via `main.js` avec un code du type :

```
store.state = 5;
```

## 14 Mode strict

[Référence : developer.mozilla.org](https://developer.mozilla.org)

L'utilisation des "modules" active le mode "strict".

```
"use strict";
```

Le code des modules est automatiquement en mode strict et aucune instruction n'est nécessaire pour passer dans ce mode.

Dans ce mode, le langage js se comporte un peu différemment :

- **Premièrement**, il est impossible de créer accidentellement des variables globales.
- **Deuxièmement**, le mode strict fait en sorte que les affectations qui échoueraient silencieusement lèveront aussi une exception. Par exemple, NaN (not a number) est une variable globale en lecture seule. En mode normal, une affectation à NaN ne fera rien ; le

développeur ne recevra aucun retour par rapport à cette faute. En mode strict, affecter une valeur quelconque à NaN lèvera une exception.

- **Troisièmement**, le mode strict lèvera une exception lors d'une tentative de suppression d'une propriété non-supprimable (là où cela ne produisait aucun effet en mode non strict) :

```
"use strict";

delete Object.prototype; // lève une TypeError
```

– **Quatrièmement**, le mode strict requiert que toutes les propriétés nommées dans un objet littéral soient uniques. En mode non-strict, les propriétés peuvent être spécifiées deux fois, JavaScript ne retenant que la dernière valeur de la propriété. Cette duplication en devient alors une source de confusion, surtout dans le cas où, dans une modification de ce même code, on se met à changer la valeur de la propriété autrement qu'en changeant la dernière instance. Les noms de propriété en double sont une erreur de syntaxe en mode strict :

```
```js
"use strict";
    var o = { p: 1, p: 2 }; // !!! erreur de
syntaxe
```

- **Cinquièmement**, le mode strict requiert que les noms de paramètres de fonction soient uniques. En mode non-strict, le dernier argument dupliqué cache les arguments précédents ayant le même nom. Ces arguments précédents demeurent disponibles via arguments[i], ils ne sont donc pas complètement inaccessibles. Pourtant, cette cachette n'a guère de sens et n'est probablement pas souhaitable (cela pourrait cacher une faute de frappe, par exemple). Donc en mode strict, les doublons de noms d'arguments sont une erreur de syntaxe :

```
function somme(a, a, c) { // !!! erreur de
syntaxe
    "use strict";
    return a + b + c; // Ce code va
planter s'il est exécuté
}
```


- ****Sixièmement****, le mode strict interdit la syntaxe octale. La syntaxe octale ne fait pas partie d'ECMAScript 5, mais elle est supportée dans tous les navigateurs en préfixant le nombre octal d'un zéro : $0644 \equiv 420$ et `"\045" \equiv "%"`. La notation octale est supportée en utilisant le préfixe `"0o"` :

```
let a = 0o10; // Notation octale ES2015
```

Les développeurs novices croient parfois qu'un zéro débutant un nombre n'a pas de signification sémantique, alors ils l'utilisent comme moyen d'aligner des colonnes de nombres mais ce faisant, ils changent la valeur du nombre ! La syntaxe octale est rarement utile et peut être utilisée de manière fautive, donc le mode strict le considère comme étant une erreur de syntaxe :

```
```js
```

```
"use strict";
```

```
var somme = 015 + // !!! erreur de syntaxe
 197 +
 142;
```

- **Septièmement**, le mode strict, à partir d'ECMAScript 2015 interdit de définir des propriétés sur des valeurs primitives. Sans mode strict, de telles définitions sont ignorées. En activant le mode strict cela lèvera une exception `TypeError`.

```
(function() {
 "use strict";
 false.true = ""; // TypeError
 (14).calvados= "maison"; //
 TypeError
 "une chaîne".de = "caractères"; //
 TypeError

})();
```

# XMLHttpRequest

## Synchrones ou Asynchrones ?

Voilà le dilemme ? Mais essayons tout d'abord de comprendre ces termes.

## Synchrones

Une requête synchrone va bloquer le déroulement de l'exécution du code jusqu'à l'obtention d'un code de réponse de la part du serveur et ce quelque soit le résultat de la requête; succès ou échec.

## ## Asynchrone

Lors d'une requête asynchrone, l'application continuera de d'exécuter votre code. Des écouteurs seront mis en place afin de détecter un changement d'état de la requête et d'agir en conséquence, comme par exemple exécuter une fonction particulière si la requête est terminée ou si cette dernière a échoué.

## ## L'objet XMLHttpRequest()

### ### Instancier XMLHttpRequest

Pour réaliser une requête HTTP, nous devons tout d'abord instancier l'objet XMLHttpRequest

```
const req = new XMLHttpRequest();
```

Jusqu'à là, ça va ? Alors continuons ...

### Ouvrir la requête

Nous allons utiliser la méthode `**open()**` de `XMLHttpRequest` afin de définir la méthode et l'url de notre requête.

```
```js
req.open('GET',
'http://www.neore.fr/mon_fichier.txt', false);
```

La méthode accepte 5 arguments dont les 2 premiers sont impératifs :

1. Le type de méthode : GET, POST, PUT, DELETE, etc
2. L'url : l'adresse absolue du fichier concerné.
3. `async` : (true ou false) afin de préciser si la requête est asynchrone ou non, elle l'est pas défaut.
4. `user` : Lors d'une connexion nécessitant une authentification, nous pouvons préciser le nom de l'utilisateur
5. `password` : Lors d'une connexion nécessitant une authentification, nous pouvons préciser le mot de passe

14.1.1 2.3 - Executer la requête

Tout est prêt, nous pouvons maintenant envoyer notre requête grâce à la méthode **send()**

```
req.send(null);
```

Mais pourquoi on met *null* en argument de cette méthode ? En voilà une bonne question ! Et bien XMLHttpRequest peut tout autant recevoir que envoyer des données. Lors de l'envoi de données, avec la méthode POST par exemple, nous passerons les données envoyées en argument de la méthode `send()`.

Dans le cas présent, nous n'envoyons aucune donnée, donc nous indiquons simplement *null*.

14.2 Traitement du résultat d'une requête

14.2.1 Les propriétés importantes de XMLHttpRequest

14.2.1.1 XMLHttpRequest.status

La propriété *status* contient le code de réponse du serveur Http correspondant à la requête que nous avons formulé. Nous pouvons noter quelques codes importants :

- 200 : Tout va bien, requête OK
- 404 : Le fichier demandé n'existe pas... un classique.
- 500 : Erreur d'exécution du serveur.. un classique lors d'un problème d'exécution de script Php par exemple

C'est cette propriété que nous interrogerons afin de connaître l'état de notre requête.

14.2.1.2 XMLHttpRequest.statusText

La propriété *statusText* contient une chaîne de caractères retournée par le serveur pour nous expliquer un peu mieux le code de résultat. Cette chaîne peut contenir des informations pertinentes pour débbugger.

14.2.1.3 XMLHttpRequest.responseText

La propriété *responseText* contient le contenu de la réponse du serveur sous la forme d'une chaîne de caractères.

14.2.2 Testons le résultat de notre requête

Notre requête est envoyée et nous attendons donc le résultat...(rappelez vous, nous avons initialisé une requête synchrone).

Une fois le résultat de la requête reçu nous pouvons tester le status de celle-ci et, si tout va bien, récupérer le contenu de la requête.

```
if (req.status === 200) {  
    // Yeah ! Super, nous avons un code de réponse  
    OK  
    // Voyons le contenu de la réponse dans la  
    console :  
    console.log(req.responseText);  
  
} else {  
    // Crénom d'un vieux mériau, y'a une .ouille  
    dans le potage !  
    // Nous n'avons pas eu un code de réponse OK,  
    mais un autre ...  
    // Voyons ça dans la console  
    console.log("Code de réponse :", req.status);  
  
    // Avec un peu plus d'info :  
    console.log("Code de réponse :",  
req.statusText);  
}
```

Nous vérifions donc que req.status est bien égal à 200. Dans le cas contraire, nous affichons le code de résultat et un peu plus d'info dans la console.

14.3 Création d'une requête asynchrone

14.3.1 Instancier XMLHttpRequest

```
const req = new XMLHttpRequest();
```

Vous voilà en terrain connu maintenant. Bravo !

14.3.2 Préparons le terrain avec quelques fonctions sympathiques

14.3.2.1 Traiter la progression de la requête : XMLHttpRequest.onprogress()

Nous allons créer une fonction qui affichera la progression de téléchargement de notre requête.

```
function maProgression(event) {  
  
    // L'argument event va contenir deux  
    propriétés intéressantes :  
    // event.loaded : nous indique la quantité de  
    d'octets téléchargés.  
    // event.total : la quantité d'octets totale  
    attendue.  
    // Nous affichons ça dans la console.  
    // MAIS, MAIS, MAIS !!! Au préalable nous  
    allons vérifier que des données  
    // existent sinon nous aurons une belle erreur  
    d'execution.
```



```
    if (event.lengthComputable) {  
        console.log("Données totales : ",  
event.total);  
        console.log("Données reçues : ",  
event.loaded);  
    } else {  
        console.log("Pas de données calculables");  
    }  
}  
  
req.onprogress = maProgression;
```

14.3.2.2 Traiter une erreur de la requête : XMLHttpRequest.onerror()

Nous allons créer une fonction qui affichera le pourquoi d'une erreur en cas d'erreur.

```
function monErreur(event) {  
    // Cette fonction sera appelée uniquement en  
cas d'erreur de la requête.  
    // Il nous suffit d'indiquer l'erreur dans la  
console pour en savoir plus.  
    console.error("Erreur", event.target.status);  
}  
  
req.onerror = monErreur; // Ceci n'est pas un  
appel direct de la fonction mais bien une
```

référence à la fonction à appeler quand l'événement se produira

14.3.2.3 Traiter le changement de statut de la requête : XMLHttpRequest.onload()

Nous allons créer une fonction qui sera exécutée à chaque changement de statut de la requête. Si nous recevons le code 200, nous pourrions afficher le contenu de la réponse.

```
function enCours(event) {  
    // On teste directement le status de notre  
    instance de XMLHttpRequest  
    if (this.status === 200) {  
        // Tout baigne, voici le contenu de la  
        réponse  
        console.log("Contenu", this.responseText);  
    } else {  
        // On y est pas encore, voici le statut  
        actuel  
        console.log("Statut actuel", this.status,  
this.statusText);  
    }  
}
```

req.onload = enCours; // Ceci n'est pas un appel direct de la fonction mais bien une référence à la fonction à appeler quand l'événement se produira

14.3.3 Ouvrir la requête asynchrone

```
req.open('GET',  
'http://www.neore.fr/mon_fichier.txt', true);
```

Nous avons donc défini le dernier argument à *true* afin de préciser que nous sommes bien en mode asynchrone.

14.3.4 Lancement de la requête

```
req.send(null);
```

- pourquoi y'a *null* comme argument ?
- Ben mon p'tit Loulou fallait pas roupiller au début du cours, allez, tu relis le début, hop, hop, hop.

14.3.5 Tester si la réponse est bien au format json avec l'objet JSON et la méthode parse

```
try {  
  JSON.parse('{}'); // {}  
  JSON.parse('true'); // true  
  JSON.parse('"toto"'); // "toto"  
  JSON.parse('[1, 5, "false"]'); // [1, 5, "false"]  
  JSON.parse('null'); // null
```

```
} catch (e) {  
console.error("Parsing error:", e);  
}
```

[cf méthode parse](#)

14.4 Exercices pratiques

14.4.1 Exercice 1 : Afficher le contenu d'une réponse dans un conteneur html

Objectif : Vous devez traiter `XMLHttpRequest.responseText` afin de remplir un conteneur *div* ayant pour *id* "ma_div".

Pour cela, vous devrez :

- Créer une div avec l'id demandé dans votre code HTML.
- Instancier `XMLHttpRequest`
- Créer une requête GET pour l'url :
http://www.neore.fr/mon_fichier.txt
- Tester le code de retour de la requête
- Traiter le contenu de la réponse
- Définir le contenu de votre div avec le contenu de la réponse

14.4.2 Exercice 2 : Envoyer des données et traiter la réponse

Objectif : Vous devrez envoyer des données avec la méthode POST et traiter XMLHttpRequest.responseText afin de remplir un conteneur *div* ayant pour *id* "ma_div2".

Pour cela, vous devrez :

- Créer une div avec l'id demandé dans votre code HTML.
- Instancier XMLHttpRequest
- Créer une requête POST pour l'url :
<http://www.neore.fr/coucou.php>
- Envoyer les données nom=votre_prenom (remplacer votre_prenom par votre vrai prénom)
- Tester le code de retour de la requête
- Traiter le contenu de la réponse
- Définir le contenu de votre div avec le contenu de la réponse

15 Promesse

[Article pour bien comprendre les promesses.](#)

L'objet Promise , apparu avec ES2015; est utilisé pour réaliser des traitements de façon asynchrone. Une promesse représente une valeur qui peut être disponible maintenant, dans le futur voire jamais !

Une promesse a 3 états :

- pending (en cours)
- resolve (résolue)
- reject (rejetée)

15.1 Ancienne méthode : via des callback

Pour ce premier exemple, j'ai choisi de ne pas utiliser de fonction asynchrone (via `setTimeout` par exemple) mais une fonction qui retourne un résultat aléatoire dans le but de simplifier le code et donc l'explication.

```
function getToken(s, f) {  
  if (Math.random() > 0.5) {  
    s("XCOE4dod340CEESee7");  
  } else f(new Error("Pas plus de token que de  
beurre à la roulante"));  
}  
  
const success = function(msg) {  
  console.log(msg);  
};  
  
const failure = function(err) {  
  console.error(err);  
};  
  
getToken(success, failure);
```

15.2 Avec les promises

La principale différence réside dans le fait que le résultat, une réussite ou un échec est renvoyé respectivement à la méthode "then" ou à la méthode "catch" :

```
getToken = () => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        const token = "qsdfEDLSoie5d8899;dEDd"
        console.log('Token ok');
        resolve(token); // renvoie le résultat à la
méthode "then()"
      } else reject(new Error("Pas de chance, vous
n'avez pas pu obtenir de token")); // renvoie le
résultat à la méthode "catch"
    }, 2000)
  })
}

getToken()
  .then(value => {
    console.log(value);
  })
  .catch(error => {
    console.error("Erreur: ", error.message);
  });
```

15.3 Le chaînage de promesses

On peut avoir besoin d'enchaîner les appels de fonction suivant le résultat d'une opération incertaine. Nous simulons ci dessous un processus dans lequel il faut d'abord obtenir un token avant de pouvoir obtenir des infos sur un utilisateur. Il faudra que la méthode "then" renvoie une autre promesse afin de pouvoir les chaîner.

```
getToken = () => {
  return new Promise((res, rej) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        const token = "qsdfeDLSoie5d8899;dEDd";
        console.log("Token ok");
        res(token);
      } else
        rej(new Error("Pas de chance, vous n'avez pas pu obtenir de token"));
    }, 2000);
  });
};

getUser = token => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (Math.random() > 0.5) {
        console.log("User ok", token);
      }
    }, 2000);
  });
};
```



```

        resolve({ id: 1, token: token });
    } else reject(new Error("Pas
d'utilisateur"));
    }, 2000);
});
};

getToken()
    .then(value => {
        console.log("value dans le premier then : ",
value);
        // notez ici que "then" doit renvoyer une
promesse pour que l'on puisse "chaîner"
        return getUser(value);
    })
    .then(value => {
        console.log("value dans le deuxième then : ",
value);
    })
    .catch(error => {
        console.error("Erreur: ", error.message);
    });

```

15.4 Async et await

La déclaration **async function** et le mot clé **await** sont des « sucres syntaxiques » apparus avec ES2017. Ils permettent de retrouver une syntaxe plus classique et donc plus lisibles.

15.4.1 async

Le mot clé **async** devant une déclaration de fonction la transforme en fonction asynchrone. Elle va retourner une promesse. Si la fonction retourne une valeur qui n'est pas une promesse, elle sera automatiquement comprise dans une promesse.

La promesse sera résolue avec la valeur renvoyée par la fonction asynchrone ou sera rompue s'il y a une exception non interceptée émise depuis la fonction asynchrone.

15.4.2 await

Le mot clé **await** est valable uniquement au sein de fonctions asynchrones définies avec **async**.

await interrompt l'exécution d'une fonction asynchrone tant qu'une promesse n'est pas résolue ou rejetée.

15.4.3 Exemple de code

Exemple 1 : Réutilisons les fonctions `getToken` et `getUser` préalablement définies avec **async et **await** :**

```
async function getTokenUser() {  
    try {  
        const token = await getToken(); // bloque  
l'exécution jusqu'à obtention de la réponse de la  
promesse
```

```

        const user = await getUser(token);
        console.log('Token et user : ', token,
user);

    } catch (error) {
        console.log('Erreur attrapée : ', error);
    }
}
getTokenUser();

```

Exemple 2 : même fonction en utilisant async et await puis then()

```

// Promesse en utilisant async et await
async function getUniversities() {
    try {
        const response = await
fetch("http://universities.hipolabs.com/search?
country=Italy");
        const universities = await response.json();
        console.log('universités : ', universities);

    } catch (error) {
        console.error('Erreur attrapée : ', error);
    }

}

// Promesse en utilisant then
function thenGetUniversities() {

```

```

    fetch("http://universities.hipolabs.com/search?
country=Italy")
    .then(response => {
        return response.json();
    })
    .then(universities => {
        console.log('universités : ', universities);
    })
    .catch(error => {
        console.error('Erreur attrapée : ', error)
    })
}

```

16 Fetch

La méthode `fetch()` permet de récupérer des ressources à travers le réseau de manière asynchrone.

Elle utilise les "promise". La réponse au "fetch" est un [objet stream](#), sur lequel on peut appeler les méthodes `json()` ou `text()` qui retournent eux-même une promesse.

Ex :

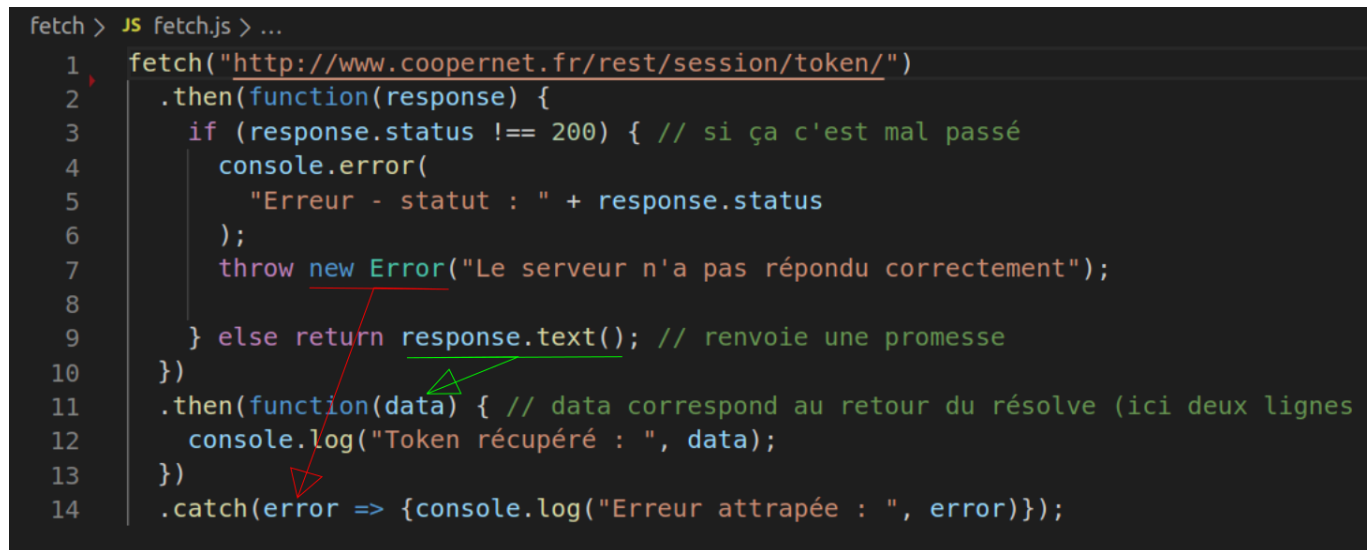
```

fetch("https://www.coopernet.fr/session/token")
    .then(function(response) {
        if (response.status === 200) { // si ça c'est
mal passé
            throw new Error("Le serveur n'a pas répondu

```

```
correctement");
    } else return response.text(); // renvoie une
promesse
  })
  .then(function(data) { // data correspond au
retour du résolve (ici deux lignes au dessus)
    console.log("Token récupéré : ", data);
  })
  .catch(error => {console.log("Erreur attrapée :
", error)});
```

L'image ci-dessous tente d'expliquer comment les données sont passées de puis la méthode "then" vers une autre méthode "then" ou vers une capture d'erreur (catch).



```
fetch > JS fetch.js > ...
1 fetch("http://www.coopernet.fr/rest/session/token/")
2 .then(function(response) {
3   if (response.status !== 200) { // si ça c'est mal passé
4     console.error(
5       "Erreur - statut : " + response.status
6     );
7     throw new Error("Le serveur n'a pas répondu correctement");
8   }
9   } else return response.text(); // renvoie une promesse
10 })
11 .then(function(data) { // data correspond au retour du résolve (ici deux lignes
12   console.log("Token récupéré : ", data);
13 })
14 .catch(error => {console.log("Erreur attrapée : ", error)});
```

16.1 Un exemple complet

```
```js
getUsers = (callbackSuccess, callbackFailed) => {
// création de la requête
```

```
console.log("Dans getUsers de coopernet.");
return fetch(this.url_server + "memo/users/", {
// permet d'accepter les cookies ?
credentials: "same-origin",
method: "GET",
headers: {
"Content-Type": "application/hal+json",
"X-CSRF-Token": this.token,
"Authorization": "Basic " + btoa(this.user.uname + ":" +
this.user.upwd) // btoa = encodage en base 64
}
})
.then(response => {
console.log("data reçues dans getUsers avant json() :",
response);
if (response.status === 200) return response.json();
else throw new Error("Problème de réponse ", response);
})
.then(data => {
console.log("data reçues dans getTerms :", data);
if (data) {
// ajout de la propriété "open" à "false" pour tous les termes
de
// niveau 1
//data.forEach()
return data;
```

```
} else {
throw new Error("Problème de data ", data);
}
})
.catch(error => { console.error("Erreur attrapée dans
getUsers", error); });
};
```

## ## Exercice 1

Utilisez le endpoint de  
"universities.hipolabs.com" dont voici un exemple  
:

[http://universities.hipolabs.com/search?  
country=Italy](http://universities.hipolabs.com/search?country=Italy)

## ### Partie 1

- Vous créez un formulaire qui permet de choisir un pays (France, Italie, Espagne, ...)
- A la validation du formulaire, vous utilisez la méthode "fetch" pour interroger l'API.
- Vous stockez les résultats dans une variable univs (array).

- Vous affichez dans l'interface le nombre de résultats.
- A partir de la variable "univs", vous créez des instances de University qui affichent les résumés d'université (Nom, site web) dans une grille bootstrap de 4 colonnes. Chaque résultat est entouré d'une bordure avec des arrondis.

### ### Partie 2

Une fois le pays choisi, un autre formulaire apparaît et permet de "filtrer" les résultats par nom. Par exemple, si vous entrez "montp", vous ne devriez voir plus que les universités dont le nom contient "montp".

- Attention à gérer les majuscules et minuscules et si vous y arrivez, les accents.
- Faites en sorte que le filtre ne commence qu'après avoir entré au moins 3 caractères
- Faites en sorte que l'on puisse "revenir" en arrière, c'est à dire par exemple, revenir à "mont" si on a entré "montk"
- Utilisez la [méthode includes]([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/St](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes)



```
ring/includes)
```

## ## Exercice 2

Inscrivez-vous sur [<https://openweathermap.org/>]  
(<https://openweathermap.org/>  
"https://openweathermap.org/").

Faites un formulaire afin d'écire le nom de la ville dont vous voulez connaître la météo.

Utilisez cette api

<https://openweathermap.org/api/geocoding-api> afin de recevoir la longitude et latitude de la ville.

Récuperez les données de cette api

<https://openweathermap.org/current> et affichez :

- L'icône.
- Le nom de la ville.
- La description du temps en français.
- Les températures actuelle, minimal et maximal, en celsius.

## ## Exercice 3 - Mise en place d'un serveur local "json-server"

## Références :

- [<https://www.npmjs.com/package/json-server>](<https://www.npmjs.com/package/json-server>)
- [<https://bobbyhadz.com/blog/npm-command-not-found-json-server>](<https://bobbyhadz.com/blog/npm-command-not-found-json-server>)

## Installer json-server

```
```shell
```

```
npm install -g json-server
```

Créer le répertoire json-server **en dehors de votre répertoire où vous allez créer votre application**

Se rendre sur le répertoire (cd json-server)

Lancer le serveur :

```
json-server --watch db.json
```

Créer un fichier db.json :

```
{  
  "tasks": [  
    { "id": 1, "label": "Faire le ménage", "done":
```

```
"false" }  
]  
}
```

Lancer la commande suivante

```
npx json-server --watch db.json --port 3000
```

Tester l'url suivante dans votre navigateur:

<http://localhost:3000/posts>

Les routes utilisables :

Requête Http	Route
--------------	-------

GET	/posts
-----	--------

GET	/posts/1
-----	----------

POST	/posts
------	--------

PUT	/posts/1
-----	----------

PATCH	/posts/1
-------	----------

DELETE	/posts/1
--------	----------

Pas besoin de préciser l'id dans le corps de votre requête

POST / PUT / PATCH

POST / PUT / PATCH doivent inclure Content-Type:
application/json dans leur header de requête.

Exemple de requête GET :

```
fetch("http://localhost:3000/posts")
  .then(response => {
    console.log(`response status`,
response.status);
    return response.json();
  })
  .then(data => {
    console.log(`data : `, data);
  })
  .catch(error => {
    console.log(`erreur attrapée : `, error);
  })
```

Exemple de requête POST :

```
fetch("http://localhost:3000/posts",
{
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  method: "POST",
  body: JSON.stringify({ "title": "Simon",
"author": "Yvan" })
})
  .then(function (res) { console.log(res) })
  .catch(function (res) { console.log(res) })
```

17 Bonnes pratiques

Le texte suivant est une traduction (maison) du [guide de Ryan McDermott](#)

Ce document s'inspire du livre Clean Code de Robert C. Martin pour l'adapter au JavaScript. C'est un guide pour produire du code lisible, réutilisable et refactorable en JavaScript.

Tous les principes énoncés ici ne doivent pas être strictement suivis et seront encore moins universellement acceptés. Ce sont des lignes directrices et rien de plus, mais ce sont celles codifiées au cours de nombreuses années d'expérience collective par les auteurs de Clean Code.

Notre métier de développeur a un peu plus de 50 ans et nous en apprenons encore beaucoup. Lorsque l'architecture logicielle sera aussi ancienne que l'architecture elle-même, nous aurons peut-être des règles plus difficiles à suivre. Pour l'instant, laissez ces directives servir de référence pour évaluer la qualité du code JavaScript que vous produisez.

Une dernière chose: sachez que ce guide ne fera pas immédiatement de vous un meilleur développeur de logiciels, et travailler avec lui pendant de nombreuses années ne signifie pas que vous ne ferez pas d'erreurs. Chaque morceau de code commence comme un premier brouillon,

comme de l'argile humide qui prend sa forme finale. Enfin, nous ciselons les imperfections lorsque nous l'examinons avec nos pairs. Ne soyez pas trop exigeant en vers vous-même pour les premières ébauches à améliorer. Concentrez vous plutôt sur votre code !

17.1 Linter

Un "linter" permet de vérifier la qualité du code. Par exemple [ESLint](#) est un outil permettant d'identifier et de signaler les pattern trouvés dans le code ECMAScript/JavaScript, dans le but de rendre le code plus cohérent et d'éviter les bugs.

[Vérifier si ESLint est activé sur Visual Studio Code.](#)

17.2 Imperative vs functional programming

17.2.1.1 Programmation fonctionnelle

Si l'on en croit wikipédia, la programmation fonctionnelle est un paradigme de programmation de type **déclaratif** qui considère le calcul en tant qu'évaluation de fonctions mathématiques.

Comme le changement d'état et la mutation des données ne peuvent pas être représentés par des évaluations de fonctions la programmation fonctionnelle ne les admet pas,

au contraire elle met en avant l'application des fonctions, contrairement au modèle de programmation impérative qui met en avant les changements d'état.

Un langage fonctionnel est donc un langage de programmation dont la syntaxe et les caractéristiques encouragent la programmation fonctionnelle.

La programmation fonctionnelle s'affranchit de façon radicale des effets secondaires (ou effets de bord) en interdisant toute opération d'affectation.

Le paradigme fonctionnel n'utilise pas de machine à états pour décrire un programme, mais un emboîtement de fonctions qui agissent comme des « boîtes noires » que l'on peut imbriquer les unes dans les autres. Chaque boîte possédant plusieurs paramètres en entrée mais une seule sortie, elle ne peut sortir qu'une seule valeur possible pour chaque n-uplet de valeurs présentées en entrée. Ainsi, les fonctions n'introduisent pas d'effets de bord. Un programme est donc une application, au sens mathématique, qui ne donne qu'un seul résultat pour chaque ensemble de valeurs en entrée. Cette façon de penser, très différente de la démarche de la programmation impérative, est l'une des causes principales de la difficulté qu'ont les programmeurs formés aux langages impératifs pour aborder la programmation fonctionnelle. Cependant, elle ne pose

généralement pas de difficultés particulières aux débutants qui n'ont jamais été exposés à des langages impératifs. Un avantage important des fonctions sans effet de bord est la facilité que l'on a à les tester unitairement. Par ailleurs, l'usage généralisé d'une gestion de mémoire automatique par l'intermédiaire d'un ramasse-miettes simplifie la tâche du programmeur.

En pratique, pour des raisons d'efficacité, et du fait que certains algorithmes s'expriment aisément avec une machine à états, certains langages fonctionnels autorisent la programmation impérative en permettant de spécifier que certaines variables sont assignables (ou mutables selon la dénomination habituelle), et donc la possibilité d'introduire localement des effets de bord. Ces langages sont regroupés sous le nom de langages fonctionnels impurs.

Les langages dits purement fonctionnels n'autorisent pas la programmation impérative. De fait, ils sont dénués d'effets de bord et protégés contre les problèmes que pose **l'exécution concurrente**.

La mise en œuvre des langages fonctionnels fait un usage sophistiqué de la pile car, afin de s'affranchir de la nécessité de stocker des données temporaires dans des tableaux, ils font largement appel à la récursivité (fait d'inclure l'appel d'une fonction dans sa propre définition). L'une des multiples

techniques pour rendre la compilation de la récursivité plus efficace est une technique dénommée récursion terminale (en anglais : tail-recursion), qui consiste à accumuler les résultats intermédiaires dans une case mémoire de la pile et à la passer en paramètre dans l'appel récursif. Ceci permet d'éviter d'empiler les appels récursifs dans la pile en les remplaçant par une simple succession de sauts. Le code généré par le compilateur est alors similaire à celui généré par une boucle en impératif.

En programmation déclarative, on décrit le quoi, c'est-à-dire le problème. Par exemple, les pages **HTML sont déclaratives** car elles décrivent ce que contient une page (texte, titres, paragraphes, etc.) et non comment les afficher (positionnement, couleurs, polices de caractères...). Alors qu'en programmation impérative (par exemple, avec le C ou Java), on décrit le comment, c'est-à-dire la structure de contrôle correspondant à la solution.

C'est une forme de programmation sans effets de bord, ayant généralement une correspondance avec la logique mathématique.

17.2.1.2 Programmation impérative

Si l'on en croit wikipédia, en informatique, la programmation impérative est un paradigme de programmation qui décrit les opérations en séquences d'instructions exécutées par

l'ordinateur pour modifier l'état du programme. Ce type de programmation est le plus répandu parmi l'ensemble des langages de programmation existants, et se différencie de la programmation déclarative (dont la programmation logique ou encore la programmation fonctionnelle sont des sous-ensembles).

17.2.1.3 Langages impératifs et processeurs

La quasi-totalité des processeurs qui équipent les ordinateurs sont de nature impérative : ils sont faits pour exécuter une suite d'instructions élémentaires, codées sous forme d'opcodes (pour operation codes). L'ensemble des opcodes forme le langage machine spécifique à l'architecture du processeur. L'état du programme à un instant donné est défini par le contenu de la mémoire centrale à cet instant.

Les langages de plus haut niveau utilisent des variables et des opérations plus complexes, mais suivent le même paradigme. Les **recettes de cuisine** et les vérifications de processus industriel sont deux exemples de concepts familiers qui s'apparentent à de la programmation impérative ; de ce point de vue, chaque étape est une instruction, et le monde physique constitue l'état modifiable. Puisque les idées de base de la programmation impérative sont à la fois conceptuellement familières et directement intégrées dans

l'architecture des microprocesseurs, la grande majorité des langages de programmation est impérative.

17.2.1.4 Instructions de la base impérative

La plupart des langages de haut niveau comporte cinq types d'instructions principales :

- la séquence d'instructions
- l'assignation ou affectation
- l'instruction conditionnelle
- la boucle
- les branchements

Séquence d'instructions

Une séquence d'instructions, (ou bloc d'instruction) désigne le fait de faire exécuter par la machine une instruction, puis une autre, etc., en séquence. Par exemple

```
\displaystyle {\mbox{ouvrirConnexion}};  
{\mbox{envoyerMessage}};{\mbox{fermerConnexion}};  
{\mbox{ouvrirConnexion}};{\mbox{envoyerMessage}};  
{\mbox{fermerConnexion}};
```

est une séquence d'instructions. Cette construction se distingue du fait d'exécuter en parallèle des instructions.

Instructions d'assignation

Les instructions d'assignation, en général, effectuent une

opération sur l'information en mémoire et y enregistrent le résultat pour un usage ultérieur. Les langages de haut niveau permettent de plus l'évaluation d'expressions complexes qui peuvent consister en une combinaison d'opérations arithmétiques et d'évaluations de fonctions et l'assignation du résultat en mémoire. Par exemple:

$$x \leftarrow 2+3;$$

assigne la valeur $2+3$, donc 5, à la variable de nom x .

Instructions conditionnelles

Les instructions conditionnelles permettent à un bloc d'instructions de n'être exécuté que si une condition prédéterminée est réalisée. Dans le cas contraire, les instructions sont ignorées et la séquence d'exécution continue à partir de l'instruction qui suit immédiatement la fin du bloc. Par exemple

$$\text{si}; \{\text{connexionOuverte}\}; \text{alors};$$

$$\{\text{envoyerMessage}\}; \{\text{si};$$

$$\{\text{connexionOuverte}\}; \text{alors};$$

$$\{\text{envoyerMessage}\}; \}$$

n'enverra le message que si la connexion est ouverte.

Instructions de bouclage

Les instructions de bouclage servent à répéter une suite d'instructions un nombre prédéfini de fois (voir Boucle_for),

ou jusqu'à ce qu'une certaine condition soit réalisée. Par exemple

```
\displaystyle tantque;\mbox{connexionNonOuverte}};alors;  
\mbox{attendreUnPeu}};tantque;  
\mbox{connexionNonOuverte}};alors;  
\mbox{attendreUnPeu}};  
bouclera jusqu'à ce que la connexion soit ouverte.
```

Il se trouve que ces quatre constructions permettent de faire tous les programmes informatiques possibles, elles permettent de faire un système Turing-complet.

Branchements sans condition

Les branchements sans condition permettent à la séquence d'exécution d'être transférée à un autre endroit du programme. Cela inclut le saut, appelé « goto » (go to, /gəʊ tu:/, « aller à ») dans de nombreux langages, et les sous-programmes, ou appels de procédures. Les instructions de bouclage peuvent être vues comme la combinaison d'un branchement conditionnel et d'un saut. Les appels à une fonction ou une procédure (donc un Sous-programme) correspondent à un saut, complété du passage de paramètres, avec un saut en retour.

Exemple de code js utilisant alternativement une approche impérative puis une approche fonctionnelle

```
// list of my friends
const friends = [
  { name: "Erwin", drinks: ["beer", "coffee"] },
  { name: "Peter", drinks: ["beer"] },
  { name: "Heidi", drinks: ["water"] }
];

// what do we want to search?
const itemToSearch = "beer";

/*****
 * imperative approach
 */

// a place to store the results
let resultImperative = [];

// go over every friend
for (friend of friends) {
  // check if the person drinks this
  if (friend.drinks.includes(itemToSearch)) {
    // add it to the results
    resultImperative.push(friend.name);
  }
}

console.log(resultImperative); // [ 'Erwin',
                                'Peter' ]

/*****
```

```
* functional approach
*/
const resultFunctional = friends
  // check if the person drinks this
  .filter(friend =>
friend.drinks.includes(itemToSearch))
  // only give me the name
  .map(friend => friend.name);
console.log(resultFunctional); // [ 'Erwin',
'Peter' ]
```