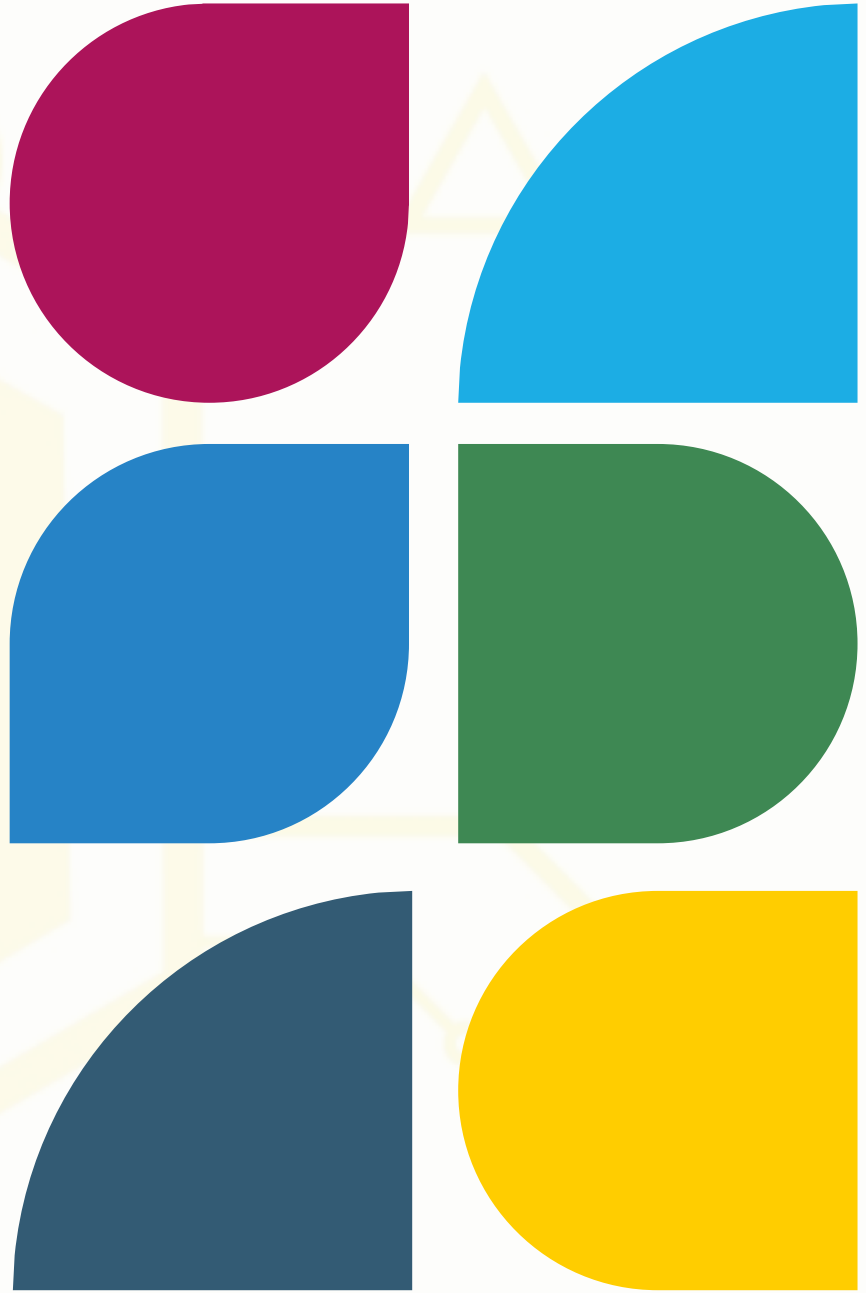
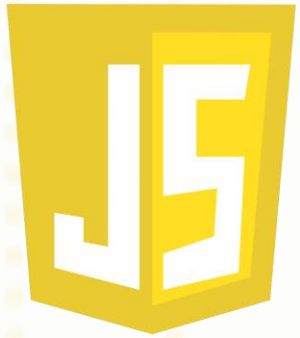
Afpa

se former, avancer



JavaScript



Coder en JavaScript..

DÉFINITION

Créé en 1995 par Brendan Eich pour la Netscape Communication Corporation.

- Le JavaScript est un langage de script basé sur la norme ECMAScript.
- Il s'insère dans le code HTML d'une page web, et permet d'en augmenter le spectre des possibilités (interactivité).
- Ce langage de POO, faiblement typé, est exécuté côté client.
- Attention : Java et Javascript sont radicalement différent.

Les caractéristiques de JavaScript :

- Pas compilé
  - Plus rapide à produire
  - Moins puissant qu'un programme en Java par exemple
- Normalisé par ECMAScript
  - https://fr.wikipedia.org/wiki/ECMAScript

OÙ SE PLACE LE CODE JAVASCRIPT ?

1. Directement dans les balises HTML
   - Utilisation du gestionnaire d'évènement :

   Un évènement qui doit déclencher le script.

   <nom eventHandler="script" /nom>

2. Entre les balises <script> </script>
   - Une nouvelle balise
     - soit dans le head (exécuté plus tard).
     - soit dans le body (à l'affichage de la page).

   Attention aux anciens navigateurs astuce

   <!-- code //>

3. Placer le code dans un fichier séparé
   - Tout comme le CSS, déclaration d'un fichier contenant le script.

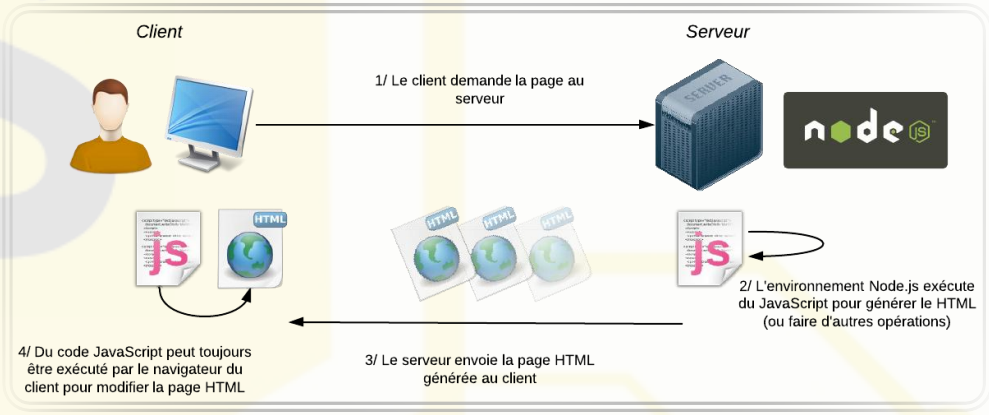
```
<!DOCTYPE html>
<html lang="fr">

    <head>

        <!-- ENCODAGE -->
        <meta charset="UTF-8">
        <!-- Comptabilité navigateur selon version -->
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <!-- description du site -->
        <meta name="description" content="Cours HTML">
        <!-- prise en charge du contexte mobile -->
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <!-- titre de la page -->
        <title>Document</title>
        <link rel="stylesheet" type="text/css" href="ressources/style.css">

        <!-- Meilleur méthode -->
        <script type="text/javascript" src="ressources/script/script.js"></
        script>
    <body>

        <!-- chargement de la page -->
        <script type="text/javascript">
            <!--
              alert('Debut du chargement de la page');
            //-->
        </script>

        <!-- directement dans la balise -->
        <a href="#" onclick="alert('Bonjour !');">lien</a>
        <a href="javascript:alert('Coucou');"> Cliquez ici </a>

        <img src="ressources/chat.gif" alt="chat" onmouseover="affichage()" />

    </body>


</html>
```

JAVASCRIPT – EXÉCUTION

JSBIN : outil en ligne permettant de tester des extraits de code en Javascript.

Le navigateur exécute automatiquement le code en suivant l'ordre d'importation dans la page web afin de manipuler le DOM.

Javascript peut s'exécuter depuis un serveur. Pour cela des environnements (tel que Node.js) permet à des applications web d'interagir entre serveur et client au travers de code écrit en Javascript.



LE DÉBOGAGE

En cas d'anomalie :

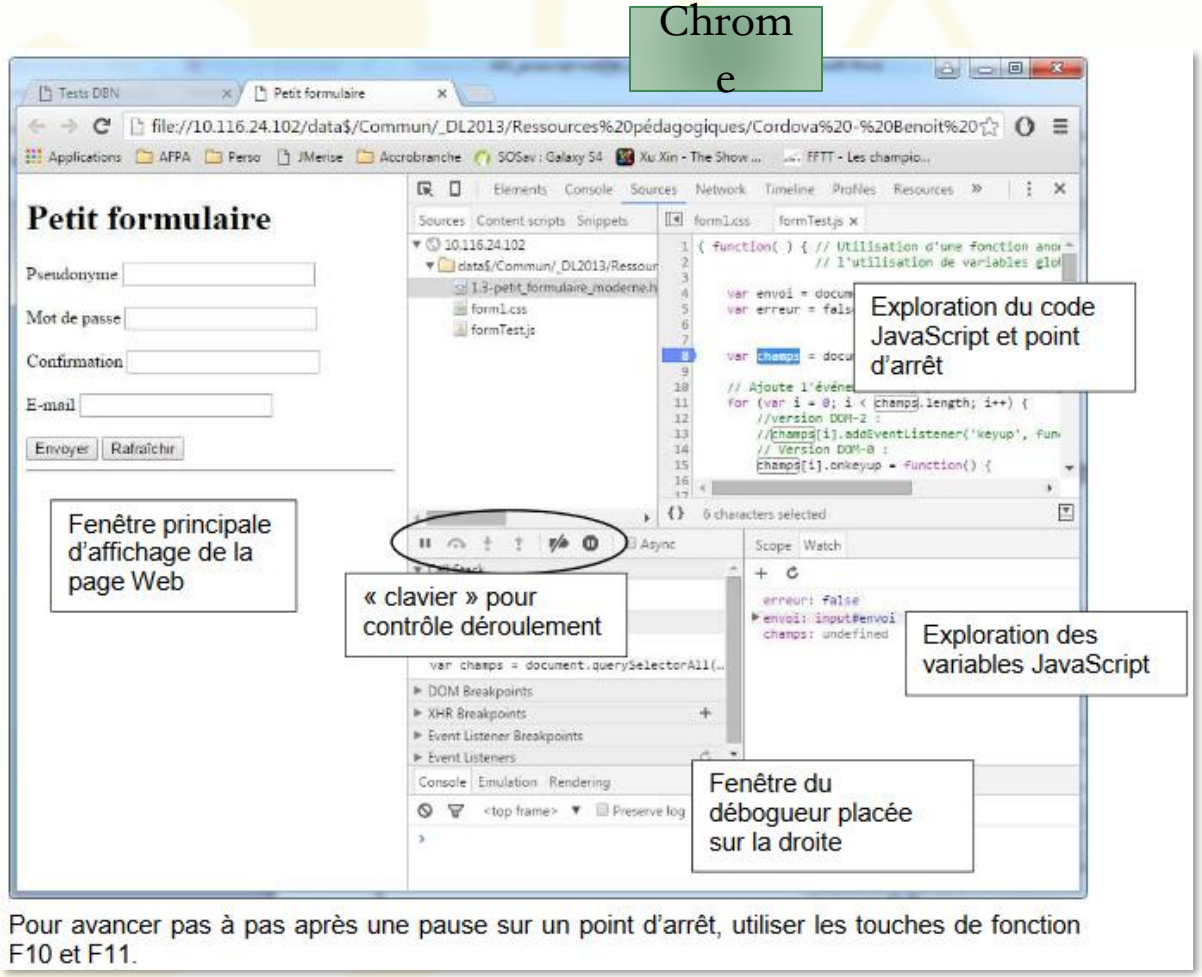
- l'interpréteur du navigateur ne va pas plus loin et se comporte comme si la page html ne contenait pas de script JavaScript.
- Une erreur de syntaxe Javascript, c'est tout le chargement du bloc qui est annulé !

L'utilisation alert() pour afficher des boites de dialogue supplémentaires permettant de tracer le déroulement d'un script.

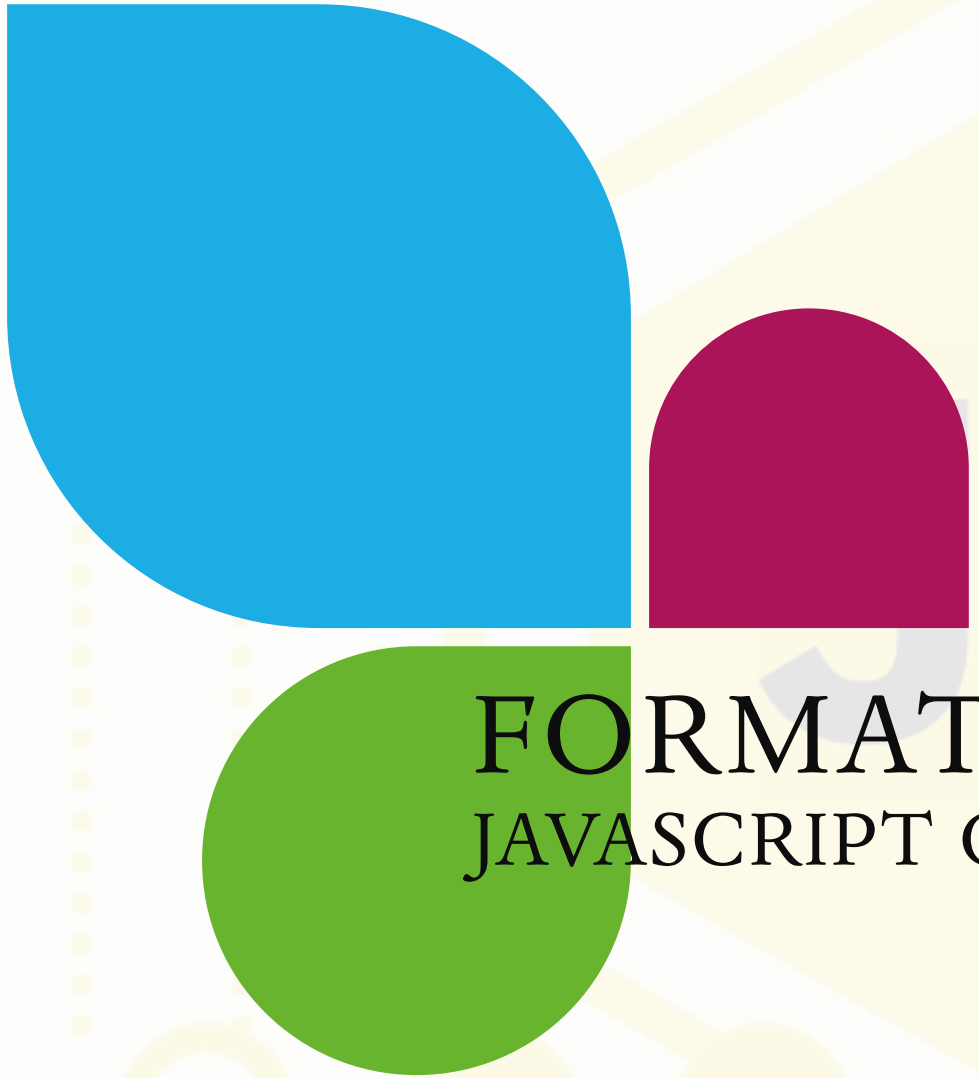
console.log() pour contrôler un ensemble de valeurs et les afficher dans la console du navigateur.

La console du navigateur est accessible en lançant l'outil de développement du navigateur.

- C'est également dans cette console que l'on pourra récupérer les informations sur les erreurs de syntaxe, effectuer des points d'arrêt, du pas à pas, …



https://fr.javascript.info/debugging-chrome



FORMAT JSON

JAVASCRIPT OBJECT NOTATION

JSON

JSON (JavaScript Object Notation) est un format de fichier textuel conçu pour l'échange de données.

Il représente des données structurées basées sur la syntaxe des objets du langage de programmation JavaScript.

De ce fait, un programme JavaScript peut convertir des données JSON en objets JavaScript natifs sans analyser ou sérialiser les données.

JSON est populaire

- en raison de son style autodescriptif,
- facile à comprendre,
- léger et compact,
- compatible avec de nombreux langages de programmation, environnements et bibliothèques.

La notation d'objets JavaScript (JSON) est un format textuel lisible par l'homme, conçu pour l'échange de données.

Il est pris en charge par de nombreux langages de programmation, environnements et bibliothèques.

JSON est remarquable car il permet aux utilisateurs de demander des données à travers les domaines en utilisant la fonction JSONP*. De plus, il est plus simple et plus léger que XML.

* méthode permettant d'envoyer des **données structurées au format JSON** entre différents domaines. L'acronyme signifie JSON (JavaScript Object **Notation**) with **P**adding (avec *formatage*).

POURQUOI L'UTILISER ?

JSON est un format qui permet de stocker des informations structurées.

Il est principalement utilisé pour transmettre les données d'une application web entre un serveur virtuel hôte et un client.

JSON apparaît dans des fichiers portant l'extension .json ou entre guillemets sous forme de chaînes de caractères ou d'objets affectés à une variable dans d'autres formats de fichiers.

JSON utilise l'analyse syntaxique côté serveur pour améliorer la réactivité.

Le processus ne nécessite aucune connaissance préalable de l'objet analysé.

C'est pourquoi JSON est largement utilisé comme format standard d'échange de données.

En outre, il permet aux utilisateurs de demander des données provenant de différents domaines en utilisant une méthode appelée JSON padding (JSONP) qui emploie des fonctions de rappel pour transmettre les données JSON.

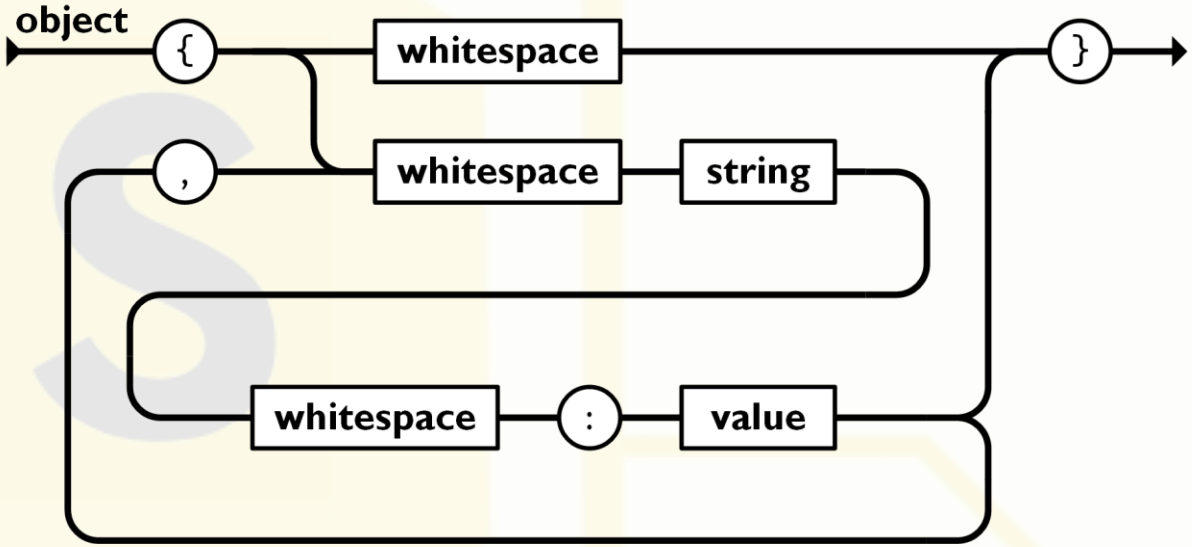
COMPRENDRE LA SYNTAXE

Comme la structure JSON est basée sur la syntaxe JavaScript object literal, ils partagent un certain nombre de similitudes.

Voici les principaux éléments de la syntaxe JSON :

- Les données sont présentées sous forme de paires clé/valeur.
- Les éléments de données sont séparés par des virgules.
- Les crochets {} désignent les objets.
- Les crochets [] désignent des tableaux.
- Par conséquent, la syntaxe des littéraux d'objets JSON ressemble à ceci :

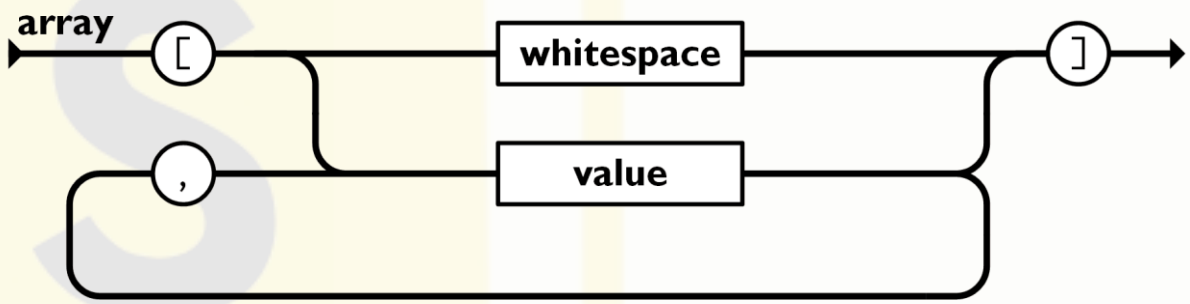
{"key":"value","key":"value","key":"value".}



LES TYPES DE VALEURS

Les tableaux :

Un tableau est une collection ordonnée de valeurs. Une valeur de tableau peut contenir des objets JSON, ce qui signifie qu'elle utilise le même concept de paire clé/valeur.




```json
{
    "students": [
        {"firstName":"Tom", "lastName":"Jackson"},
        {"firstName":"Linda", "lastName":"Garner"},
        {"firstName":"Adam", "lastName":"Cooper"}
    ]
}
```

LES TYPES DE VALEURS

Les chaînes de caractères : Les valeurs de type chaîne sont des séquences définies de zéro ou plusieurs caractères Unicode, entourées de guillemets.

{ "firstName" : "Tom" }

Nombre : Un nombre dans JSON doit être un nombre entier ou un nombre à virgule flottante.

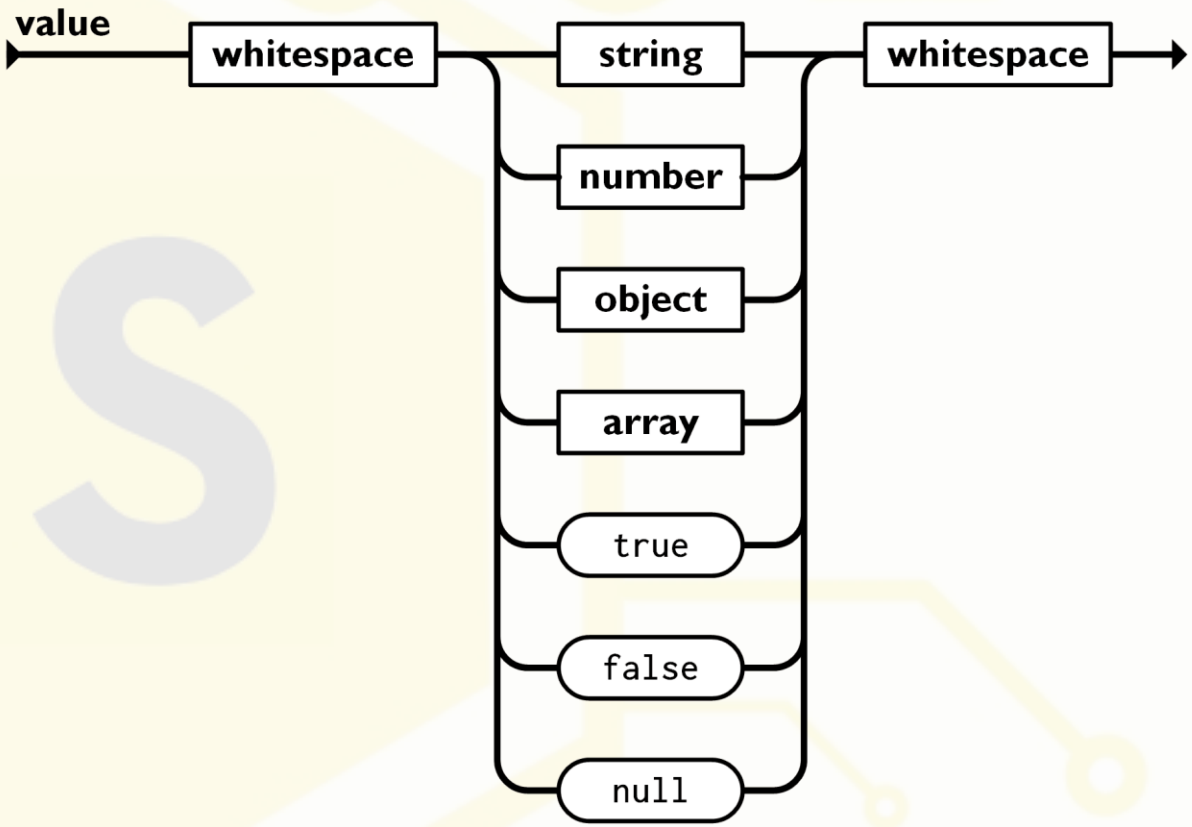
{ “age” : 30 }

Booléen : Les booléens contiennent les valeurs vrai ou faux.

{ “married” : false }

Null : Null est une valeur vide. C’est pour montrer qu’il n’y a pas d’informations.

{ "bloodType" : null }



LES OBJETS JSON

Les objets JSON sont constitués de paires de deux composants :

- Les clés sont des chaînes de caractères – des séquences de caractères entourées de guillemets.
- Les valeurs sont des types de données JSON valides. Elles peuvent se présenter sous la forme d'un tableau, d'un objet, d'une chaîne de caractères, d'un booléen, d'un nombre ou de null.
- Un deux-points est placé entre chaque clé et chaque valeur, et une virgule sépare les paires. Les deux éléments sont placés entre guillemets.

Ici, "employees" est la clé, tandis que tout ce qui se trouve entre accolades est l'objet.

MÉTHODE DE STOCKAGE JSON

Il existe deux façons de stocker des données JSON : les objets et les tableaux.

- Utiliser les tableaux

Une autre façon de stocker des données consiste à utiliser des tableaux. Les valeurs sont placées entre crochets, et des virgules séparent chaque ligne. Chaque valeur des tableaux JSON peut être d'un type différent.

Les tableaux peuvent être utiles lorsqu'ils sont associés pour surmonter le problème de l'inter-domaine.

Ils prennent également en charge la boucle qui permet aux utilisateurs d'exécuter des commandes répétées pour rechercher des données, rendant le processus plus rapide et plus efficace.

EXEMPLE

Voici un exemple simple d'utilisation de JSON, Voici ce que chaque paire indique :

- La première ligne « className » : « Class 2B » est une chaîne de caractères.
- La deuxième paire « year »:2022 a une valeur numérique.
- La troisième paire « phoneNumber »:null représente un null – il n'y a pas de valeur.
- La quatrième paire « active »:true est une expression booléenne.
- La cinquième ligne « homeroomTeacher »: {« firstName » : « Richard », « lastName » : « Roe »} représente un objet littéral.
- Enfin, le script à partir de la sixième ligne est un tableau.

```
{
    "className":"Class 2B",
    "year":2022,
    "phoneNumber":null,
    "active":true,
    "homeroomTeacher": {
        "firstName":"Richard", "lastName":"Roe"
    },
    "members": [
        { "firstName":"Jane","lastName":"Doe"},
        {"firstName":"Jinny","lastName":"Roe"},
        {"firstName":"Johnny","lastName":"Roe"}
        ]
}
```

LE LANGAGE

DÉCOUVERTE

HTTPS://WWW.W3SCHOOLS.COM/JS/
HTTPS://FR.JAVASCRIPT.INFO/
HTTPS://DEVELOPER.MOZILLA.ORG/FR/DOCS/WEB/JAVASCRIPT

DÉCOUVERTE DU LANGAGE

La base

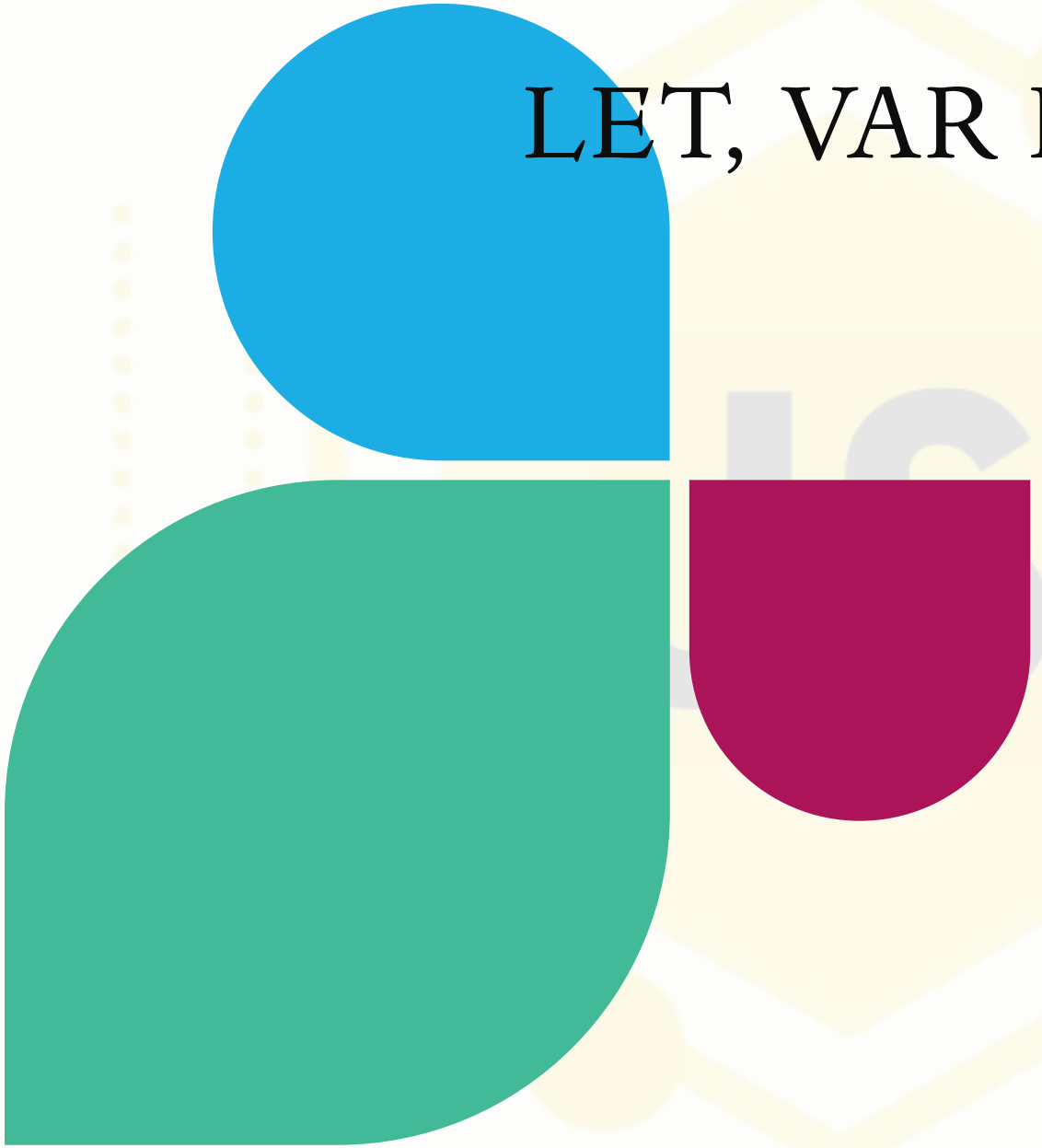
1. Sensible à la case : alert() et non Alert()
2. // commentaire ou /* … */
3. ; pour terminer une instruction. Même si c'est possible sans, cela permet d'éviter les erreurs.
4. Pour les anciens navigateurs, on doit encapsuler notre code entre les balises de commentaires HTML

```
<!-- chargement de la page -->
<script type="text/javascript">
    <!--
      alert('Debut du chargement de la page');
    //-->
</script>
```

Les variables

- soit avec le mot clé var
  - *Attention : vous pourrez croiser le mot clé var plutôt que let. Pour l'instant considérer var comme l'ancienne version de let*
  - let variableA, variableB, … ;
  - let number = 1;
  - **Typée dynamiquement et à typage faible.**
  - **Mutabilité des variables.**
  - Une variable non déclarée est **undefined**
- const pour une constante

LET, VAR ET CONST



LET ET VAR ET CONST

Javascript a été créé à l'origine avec un seul mot-clé pour définir une variable, ce mot réservé est : var.

Mais depuis la version 6 (ECMAScript 2015) de la spécification du langage, deux nouveaux mot-clés sont apparus : let et const.

Si l'on devait résumer la définition de ces deux nouvelles manières de déclarer des variables, on pourrait dire que :

- let sert à définir une variable locale
- const sert à déclarer une référence constante

Attention, une référence constante ne veut pas dire que la valeur derrière la référence est "immutable", mais que la référence elle-même est immutable.

Mais voyons exactement en quoi le fonctionnement des variables var, let et const sont différents dans la pratique :

- Stocké en global ?
  - **var** : oui ✔
  - **let** : non ❌
  - **const** : non ❌

Lorsque var est utilisé pour déclarer une variable en dehors d'une fonction, alors cette variable sera forcément référencée dans l'objet global du script.

- Se limite à la portée d'une fonction ?
  - **var** : oui ✔
  - **let** : oui ✔
  - **Const** : oui ✔

La portée (ou scope) d'une fonction est la seule à mettre toutes les variables sur un même pied d'égalité et déclarer une variable à l'intérieur d'une fonction n'aura logiquement pas d'incidence sur le reste des données du code.

LET ET VAR ET CONST

Se limite à la portée d'un block ?

- var : non ❌
- let : oui ✔️
- const : oui ✔️

Un bloc d'instruction se trouve souvent après un if, else, for, while, etc.

Donc souvenez-vous qu'une variable déclarée avec le mot clé var dans un for sera automatiquement globale.

Sauf si le for en question est contenu dans une fonction, évidemment.

Peut être réassigné ? Peut être redéclaré ?

Peut être réassigné ?

- var : oui ✔️
- let : oui ✔️
- const : non ❌

Peut être redéclaré ?

- var : oui ✔️
- let : non ❌
- const : non ❌

LET ET VAR ET CONST

Est affecté par le hoisting ?

- var : oui ✔
- let : non ❌
- const : non ❌

Le hoisting :

- Cette mécanique consiste donc à faire "virtuellement" remonter la déclaration d'une variable (ou d'une fonction) tout en haut de son scope lors de l'analyse du code par le moteur d'interprétation Javascript.
- *C'est une mécanique automatique et obligatoire qui fait partie de la spécification ECMAScript même si le terme "hoisting" n'y apparait pas en tant que tel.*

```
var x = 3;
console.log(x); /* 3 */
```

→

```
var x;
x = 3;
console.log(x); /* 3 */
```

```
console.log(x);
var x = 3;
```

→

```
var x;
console.log(x);
x = 3;
```

undefined

```
f();
function f() { console.log("hello");}
```

→

```
function f() { console.log("hello");}
f();
```

```
f();
var f = function() { console.log("hello");}
```

→ F is not function

SCOPE DES VARIABLES

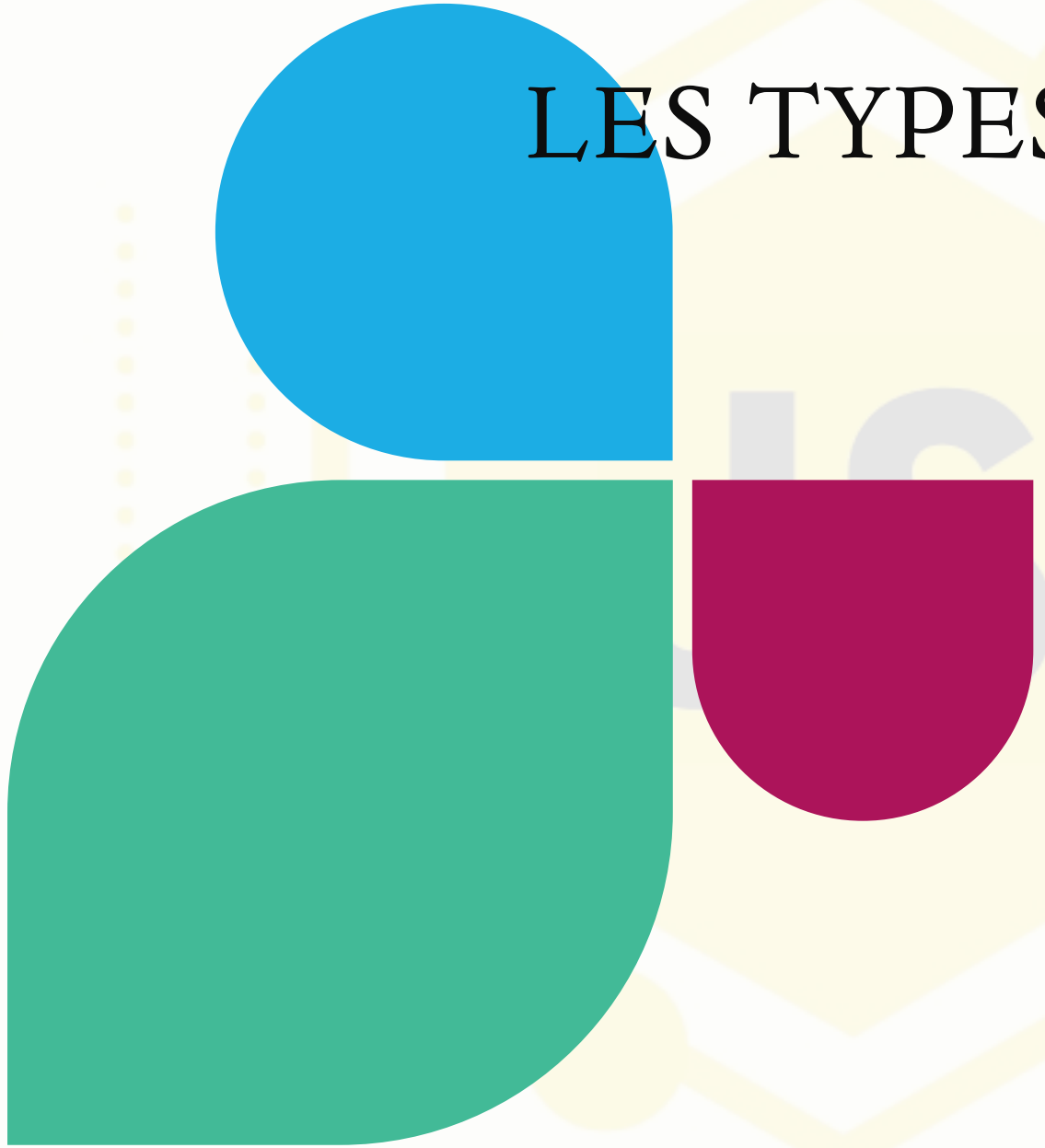
les variables créées par let ou const ne peuvent être vues ou utilisées qu'à l'intérieur du bloc de code entre accolades {} dans lequel elles sont déclarées.

Une affectation à une variable non déclarée la crée implicitement en tant que variable globale.

- a = 100 équivalents à var a = 100;
- Les variables créées avec var ont un comportement différent. Javascript la considère comme global même si elle est déclarée dans le scope d'une fonction.

```
// 1er cas
let a;              // Declaration
a = 100;            // Assignment
console.log(a);     // Usage

// 2eme cas : Hoisting
function codeHoist(){
    a = 10;         // global scope
    let b = 50;
}
codeHoist();
console.log(a);     // 10
console.log(b);     // ReferenceError : b is not defined

// 3eme cas : var code (global)
console.log(nom);   // undefined
var nom = 'Mukul Latiyan';
// how interpreter sees the above code
var nom;
console.log(nom); // undefined
nom = 'Mukul Latiyan';

// 4eme cas : Function scoped
function fun(){
    console.log(nom);
    var nom = 'Mukul Latiyan';
}
fun(); // Undefined
// idem
function fun(){
    var name;
    console.log(name);
    name = 'Mukul Latiyan';
}
fun(); // undefined
```

LES TYPES



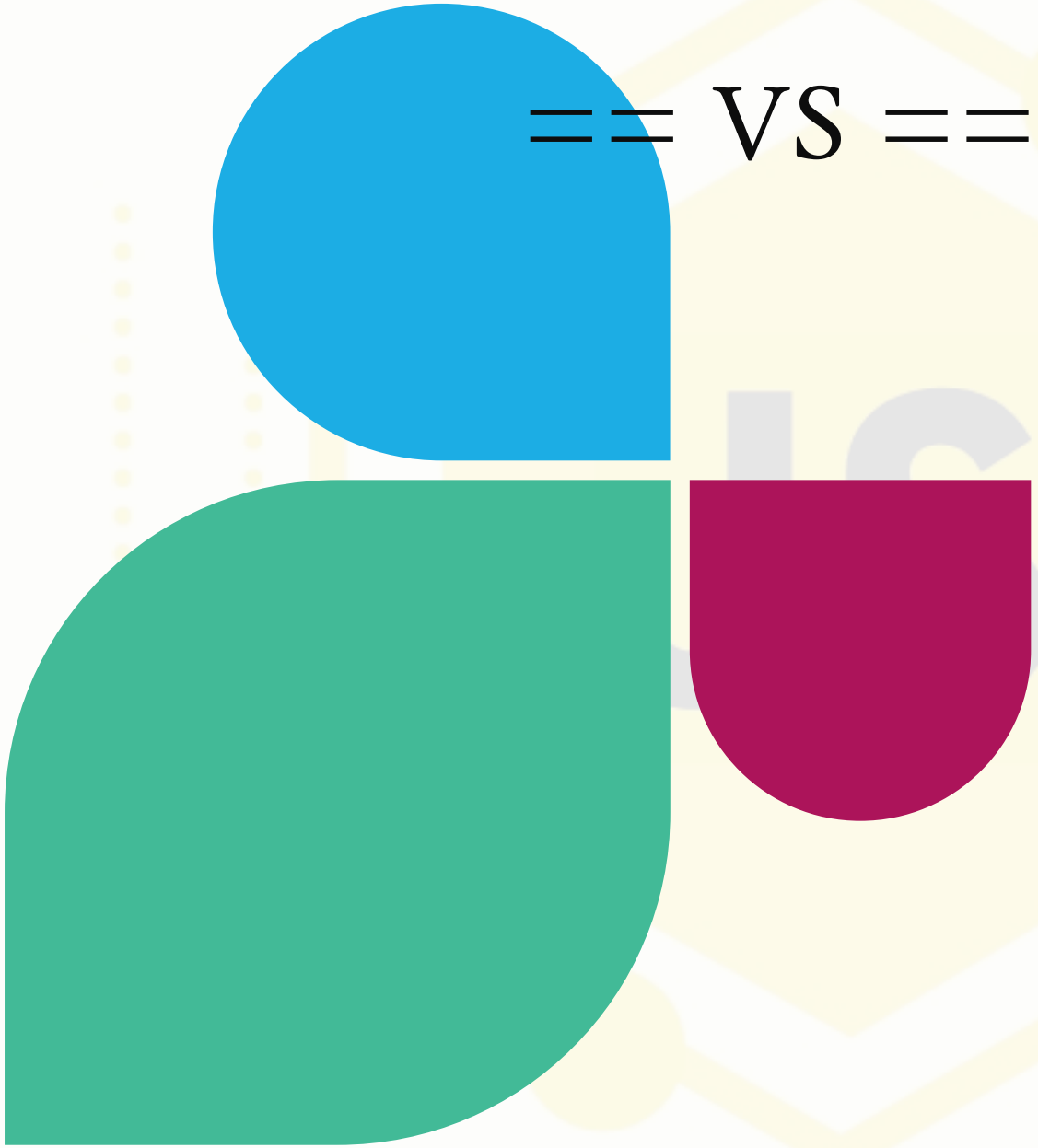
LES TYPES

En javascript, on appelle ces types, des valeurs primitives, fondamentaux qui ne sont pas des objets.

- Cependant, il est important de noter que JavaScript traite parfois les primitifs comme des objets temporaires pour permettre l'accès aux méthodes et propriétés

- Les valeurs primitives :
  - Le type booléen
  - Le type nul (null)
  - Le type indéfini (undefined)
  - Le type nombre
  - Le type pour les chaînes de caractères
  - Le type symbole Symbol("name")
- Les objets (des ensembles de propriétés).
- typeof() permet de vérifier le type en cours.

== VS ===



== VS ===

== correspond à une comparaison d'égalité abstraite

=== correspond à une comparaison d'égalité stricte

Il vaut mieux privilégier l'utilisation de l'égalité stricte

On considère que ce n'est jamais une bonne idée d'utiliser l'égalité faible.

- Le résultat d'une comparaison utilisant l'égalité stricte est plus simple à appréhender et à prédire, de plus il n'y a aucune conversion implicite ce qui rend le test plus rapide.

- L'égalité faible (==) effectuera une conversion des deux éléments à comparer avant d'effectuer la comparaison
- l'égalité stricte (===) effectuera la même comparaison mais sans conversion préalable (elle renverra toujours false si les types des deux valeurs comparées sont différents)
- https://developer.mozilla.org/fr/docs/Web/JavaScript/Equality_comparisons_and_sameness

```js
var num = 0;
var obj = new String("0");
var str = "0";

console.log(num === num); // true
console.log(obj === obj); // true
console.log(str === str); // true

console.log(num === obj); // false
console.log(num === str); // false
console.log(obj === str); // false
console.log(null === undefined); // false
console.log(obj === null); // false
console.log(obj === undefined); // false
```

```js
var num = 0;
var obj = new String("0");
var str = "0";

console.log(num == num); // true
console.log(obj == obj); // true
console.log(str == str); // true

console.log(num == obj); // true
console.log(num == str); // true
console.log(obj == str); // true
console.log(null == undefined); // true

// Les deux assertions qui suivent sont fausses
// sauf dans certains cas exceptionnels
console.log(obj == null);
console.log(obj == undefined);
```

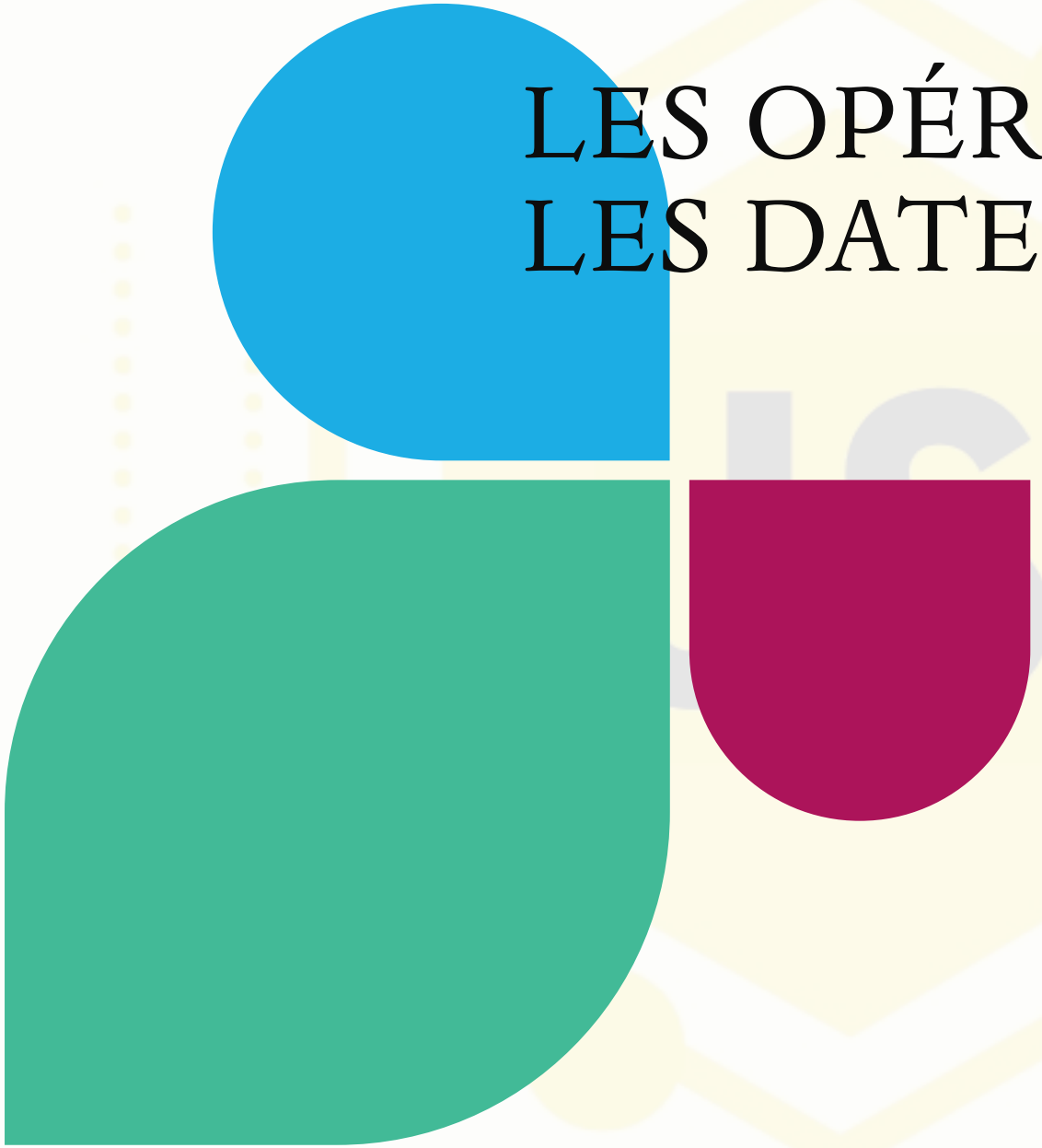
L'ÉGALITÉ FAIBLE AVEC ==

Le test d'égalité faible compare deux valeurs après les avoir converties en valeurs d'un même type.

- Une fois converties (la conversion peut s'effectuer pour l'une ou les deux valeurs), la comparaison finale est la même que celle effectuée par ===.
- L'égalité faible est symétrique : A == B aura toujours la même signification que B == A pour toute valeur de A et B.
- ToNumber(A) correspond à une tentative de convertir l'argument en un nombre avant la comparaison.
- ToPrimitive(A) correspond à une tentative de convertir l'argument en une valeur primitive grâce à plusieurs méthodes comme A.toString et A.valueOf.

		Opérande B					
		Undefined	Null	Number	String	Boolean	Object
Opérande A	Undefined	true	true	false	false	false	false
	Null	true	true	false	false	false	false
	Number	false	false	A === B	A === ToNumber(B)	A === ToNumber(B)	A == ToPrimitive(B)
	String	false	false	ToNumber(A) === B	A === B	ToNumber(A) === ToNumber(B)	A == ToPrimitive(B)
	Boolean	false	false	ToNumber(A) === B	ToNumber(A) === ToNumber(B)	A === B	false
	Object	false	false	ToPrimitive(A) == B	ToPrimitive(A) == B	ToPrimitive(A) == ToNumber(B)	A === B

LES OPÉRATEURS ET LES DATES



DÉCOUVERTE DU LANGAGE

Les opérateurs

Opérateurs	Descriptions
+, -, *, /, %, =	Opérateurs arithmétiques de base
==, ===, <, >, <=, >=, !=, !==	Opérateurs de comparaison
+=, -=, *=, /=, %=	Opérateurs associatifs
&&, ||, !	Opérateurs logiques (AND, OR et NOT)
X++, x--	Opérateurs d'incrémentation et de décrémentation
+	Concaténation de chaînes de caractères

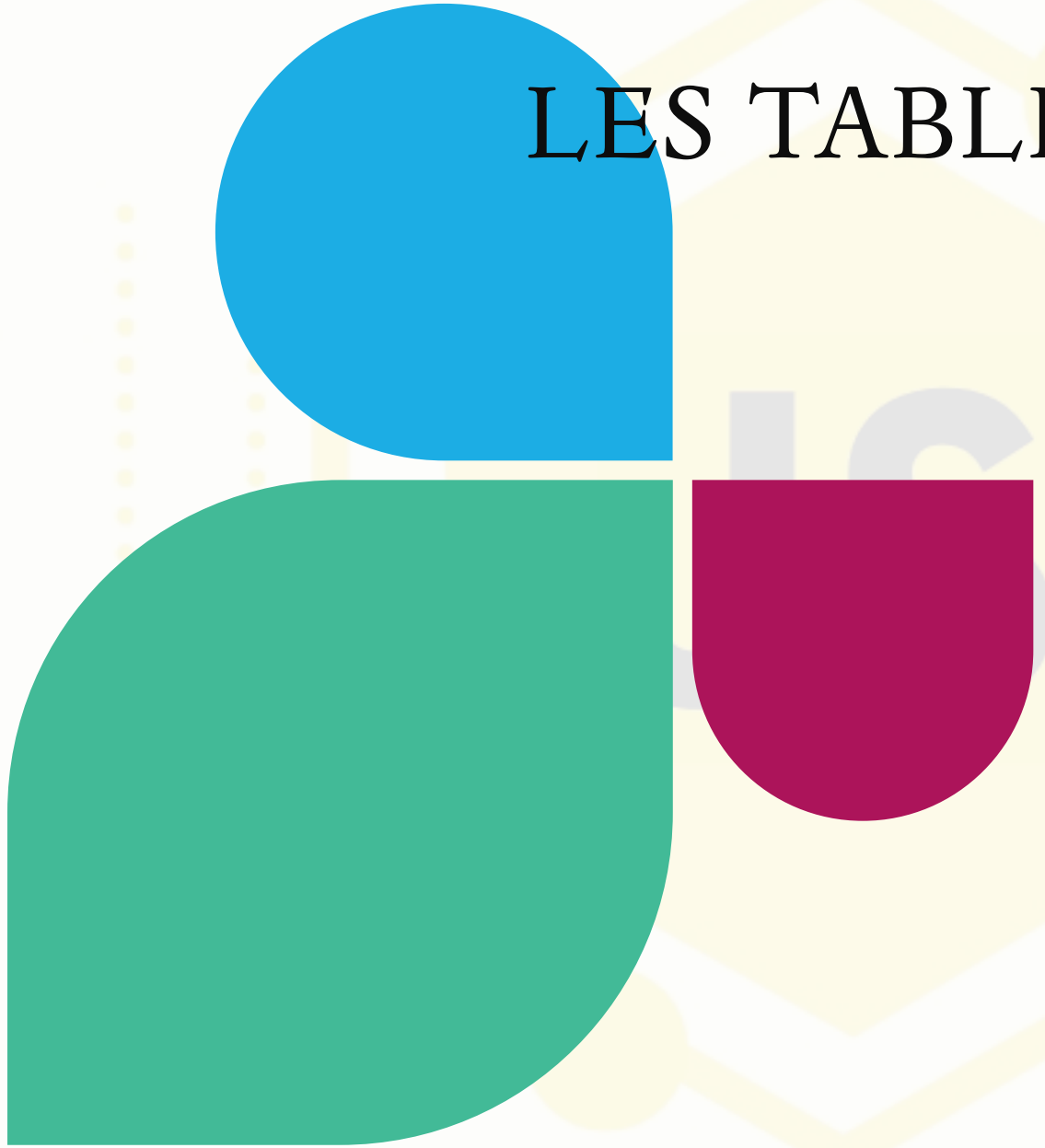
Les dates

L'objet Date de JavaScript peut être utilisé pour obtenir l'année, le mois et le jour.

- Date() : let myDate = new Date();

L'objet s'accompagne d'un ensemble de méthodes pour obtenir, modifier, afficher une date.

LES TABLEAUX

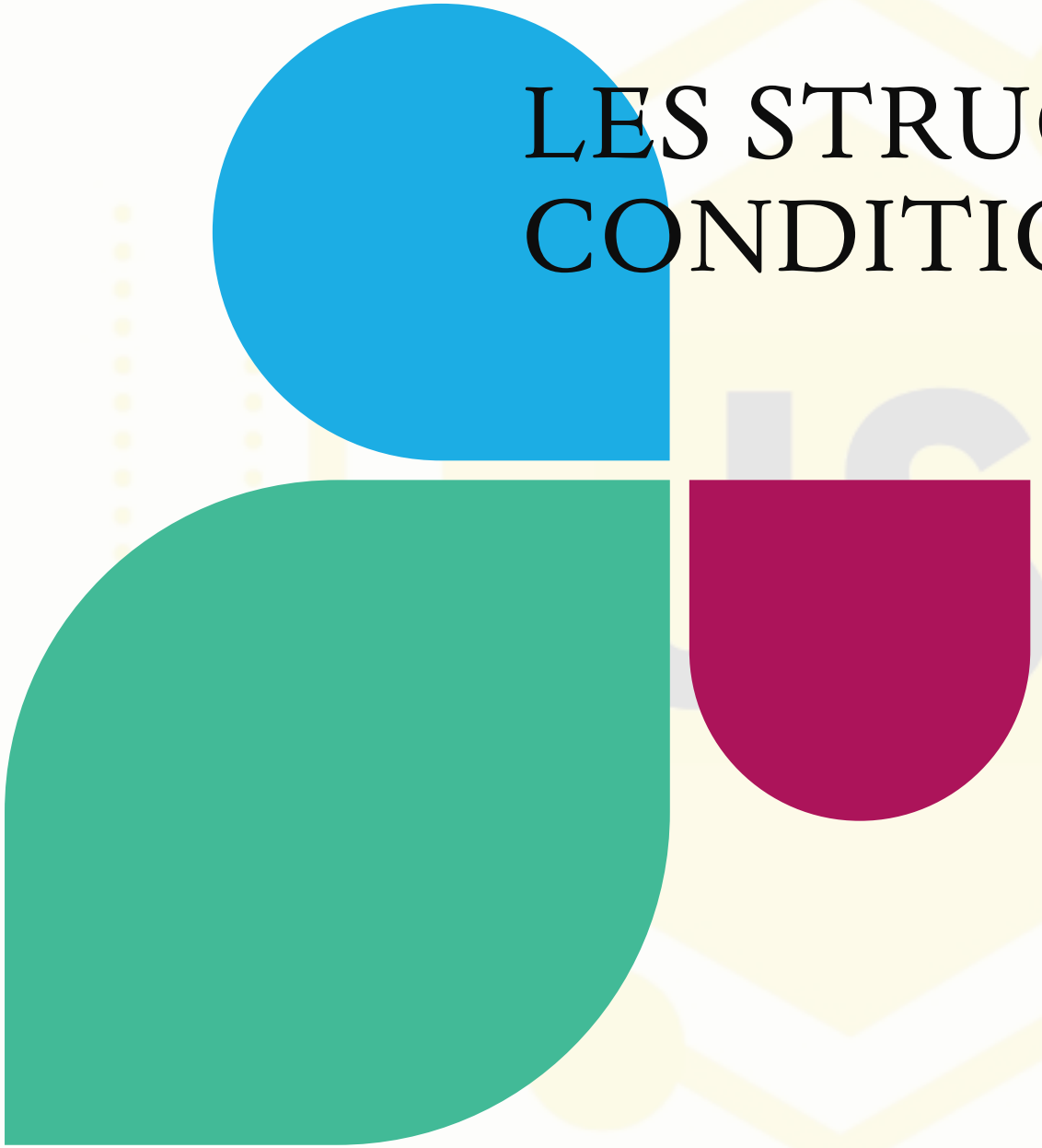


LES TABLEAUX

En Javascript, nous avons la possibilité de créer des tableaux

- let tab=[valeur1, …, valeurN];
- let tab=new Array();
  - indice de départ est 0
  - les tableaux utilisent le passage par référence
- les tableaux sont accompagnés par de nombreuses méthodes :
  - find() : recherche
  - pop() : suppression
  - push() : ajout
  - shift() : supprime le 1er élement
  - Etc…
- Pour parcourir un tableau :
  - tab.forEach(num -> console.log(num));

LES STRUCTURES CONDITIONNELLES



DÉCOUVERTE DU LANGAGE

Les conditions

```
// l'expression if
if (condition) une_instruction;

if (condition) {
    instruction1;
} else if (autre_condition) {
    instruction2;
} else {
    instruction3;
}

// ternaire
(test_condition) ? valeur_vrai  : valeur_faux;
```

Switch

```
// Switch
var animal = "oiseau";
switch(animal) {
    case "chien": ....
    case "oiseau" : ....
    case "poisson": ....
    case "vache" :
        console.log("C'est un vertébré");
    break;
    case "mouche" :....
    default :
        console.log("C'est un invertébré");
}
```

DÉCOUVERTE DU LANGAGE

Les répétitions

```
// Les répétitions
for (var i=0; i<100; i++) {
    console.log("Prèfère la boucle for si tu connais le nombre !");
}

var i;
while (!i) {
    i = confirm("As-tu compris ?");
}

do { // l'instruction suivante sera exécutée au moins 1 fois !
    i = prompt("laisse vide ou annule");
} while (i);

break;      // pour arrêter une boucle for ou while
continue;   // pour sauter une instruction ou passer à l'itération suivante
```

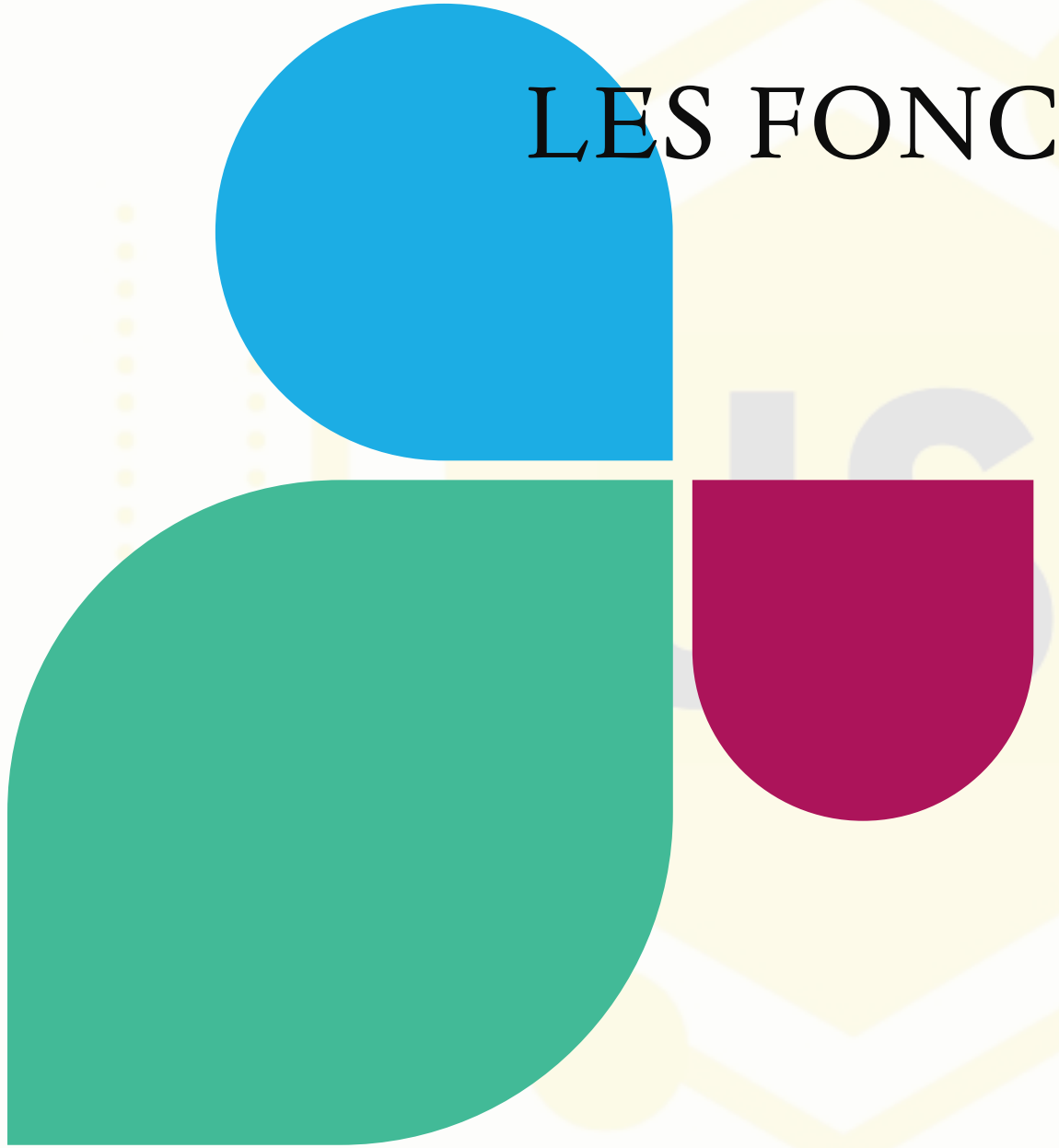
Parcours de tableaux

```
const passengers = [
    "Will Alexander",
    "Sarah Kate'",
    "Audrey Simon",
    "Tao Perkington"
];

for (let i in passengers) {
    console.log("Embarquement du passager " + passengers[i]);
}

const list = [
    "Will Alexander",
    "Sarah Kate",
    "Audrey Simon",
    "Tao Perkington"
];

for (let passenger of passengers) {
    console.log("Embarquement du passager " + passenger);
}
```

avec l'indice

avec les éléments du tableau

LES FONCTIONS



LES FONCTIONS

Définie dans un bloc d'instruction { } à un seul endroit du script, réutilisable et exécutable par un simple appel depuis le script ou depuis une autre fonction.

Un mot-clé function permettant de définir ses propres fonctions.

Deux types de fonctions :

- Avec résultat → utilisation d'un return
- Sans résultat → la fonction devient une procédure.

```
function nom([param[, param[, ... param]]]) {
  instructions
}
```

JavaScript regorge de fonctions natives :

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

Par exemple :

- isNaN(x) true si le paramètre x n'est pas un nombre.
- parseFloat(string) convertit la chaîne en nombre à virgule flottante.
- parseInt(string) convertit la chaîne en entier.

Il existe plusieurs sortent de fonctions :

- Les instructions de fonctions (les plus courantes),
- Les expressions de fonctions,
- Les fonctions anonymes (qui servent à isoler une partie du code),
- Les fonctions « Callback »,
- Les fonctions auto-exécutables,
- Les fonctions issues d'un objet (les méthodes et les constructeurs).

LES FONCTIONS

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Functions#lobjet_arguments

Instructions de fonctions

Syntaxe :

Function myName (paramètres) { instructions };

Appel :

myName(paramètres);

Exemples :

```
function coucou() { alert("coucou"); }

function calculeSurface(largeur, hauteur) {
    return largeur * hauteur;
}
```

Les paramètres des fonctions

Plus souple, JavaScript n'impose pas de respect strict des paramètres attendus.

A chaque appel d'une fonction, l'objet arguments stocke tous les paramètres envoyés.

C'est au développeur de faire en sorte de traiter la surcharge de la méthode

```
function perimetre(largeur, longueur) {
    var res = 0;
    // test si au moins un paramètre reçu
    if (!largeur) res = 0;
    // 1 param recu : carré
    else if (!longueur) { res = 4*largeur; }
    // 2 param : rectangle
    else if (arguments.length == 2) { res = (largeur + longueur)*2; }
    // polygone
    else { for (i in arguments) res += arguments[i]; }
    console.log(resultat);
}
```

PARAMÈTRES DES FONCTIONS

valeur par défaut

On a la possibilité d'indiquer une valeur par défaut pour nos paramètres.

```
function multiply(a, b = 1) {
  return a * b;
}

console.log(multiply(5, 2));
// Expected output: 10

console.log(multiply(5));
// Expected output: 5
```

Rest Parameters

Cela permet de représenter un nombre indéfini d'arguments sous forme d'un tableau.

```
function sum(...theArgs) {
  let total = 0;
  for (const arg of theArgs) {
    total += arg;
  }
  return total;
}

console.log(sum(1, 2, 3));
// Expected output: 6

console.log(sum(1, 2, 3, 4));
// Expected output: 10
```

LES FONCTIONS

Les expressions de fonctions

Les expressions de fonctions passent par la création d'une variable affectée par la définition d'une fonction.

- peuvent être écrite n'importe où dans le code
- Ne peuvent pas être appelées avant d'avoir été déclarées.

```
let surface = function calculeSurface(largeur, hauteur) {
    return largeur * hauteur;
}
```

```
// utilisation de la variable
surface(4,6);
```

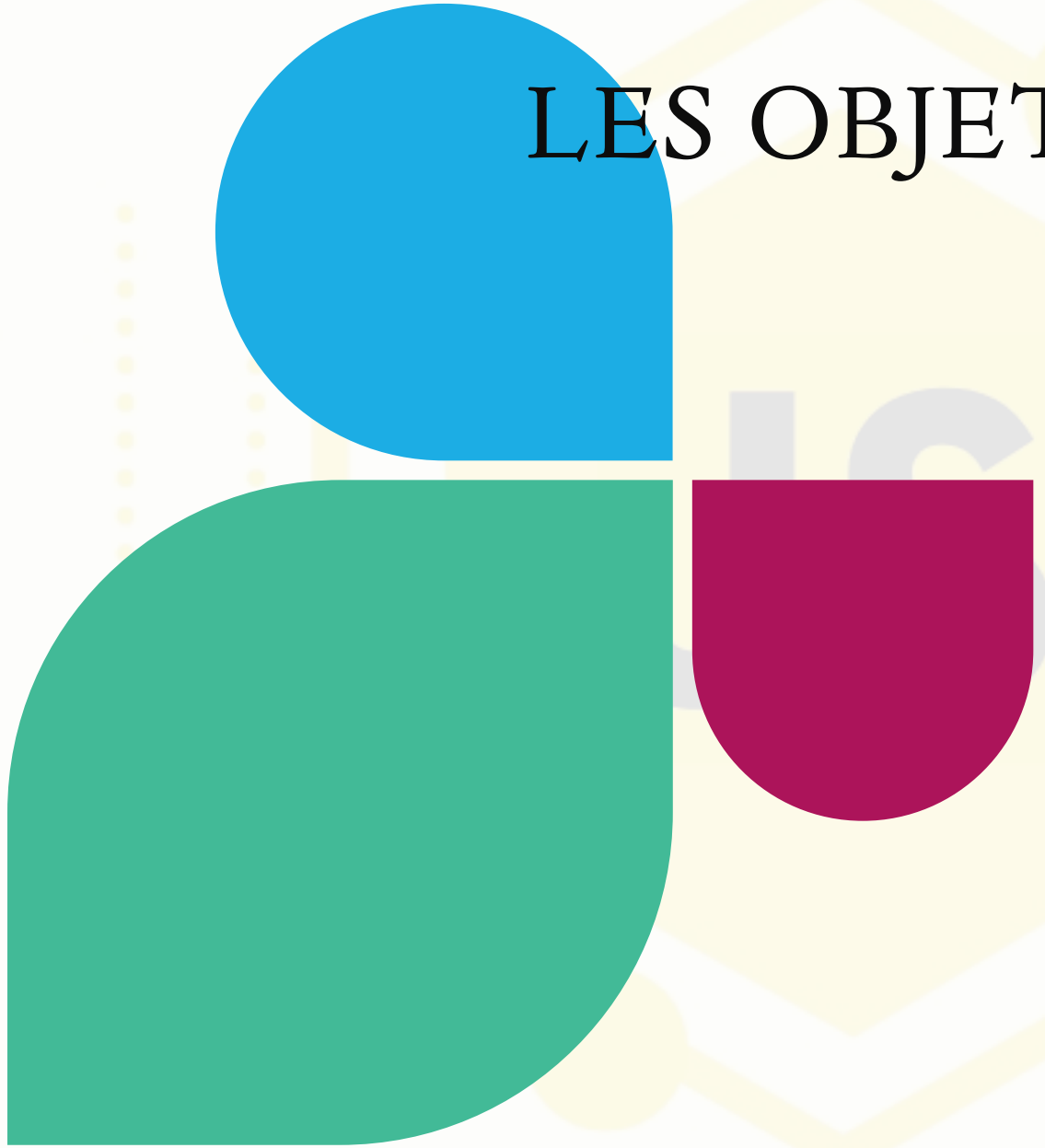
Fonctions anonymes

La fonction ne porte pas de nom. On dit qu'elle est anonyme.

- Très utilisées notamment dans la gestion d'évènements, les objets, les closures et les callback…
- Surcharge le code au détriment de sa lisibilité

```
// se lance toutes les secondes
var decompte = setInterval(function() {
        // décompte de 10 à 1
        console.log(i--);
    }, 1000);
```

LES OBJETS



LES OBJETS

Les objets

Les objets Javascripts sont écrits en JSON par des séries de paires clés-valeurs séparées par des virgules, entre des accolades.

Les objets peuvent être enregistrés dans une variable :

```
let myBook = {
    title: 'The Story of Tau',
    author: 'Will Alexander',
    numberOfPages: 250,
    isAvailable: true
};
```

Accès à un objet

Pour accéder à un attribut d'un objet :

```
let title = myBook.title;
let author = myBook["author"];
```

FREEZE ET SEAL

Protéger les objets avec freeze et seal.

```
/*
  Object.freeze
  - L'objet est complètement bloqué.
  - Impossible de modifier, ajouter ou supprimer des propriétés.
*/
const hero = { name: 'Superman', power: 'Flight' };
Object.freeze(hero);

hero.power = 'Invisibility'; // ❌ Pas possible
hero.city = 'Metropolis'; // ❌ Pas possible
delete hero.name; // ❌ Pas possible

console.log(hero); // { name: 'Superman', power: 'Flight' }


/*
  Object.seal
  - L'objet est scellé.
  - Impossible d'ajouter ou de supprimer des propriétés.
  - Les propriétés existantes peuvent être modifiées.
*/
const hero2 = { name: 'Thor', power: 'Lightning' };
Object.seal(hero2);

hero2.power = 'Stormbreaker'; // ✅ Possible
hero2.weapon = 'Mjolnir'; // ❌ Pas possible (ajout impossible)
delete hero2.name; // ❌ Pas possible

console.log(hero2); // { name: 'Thor', power: 'Stormbreaker' }

```

LA NOTION DE PROTOTYPE EN JAVASCRIPT

LES PROTOTYPES

JavaScript est un langage basé sur un prototype, ce qui signifie que les propriétés et les méthodes des objets peuvent être partagées par des objets généralisés qui ont la capacité d'être clonés et étendus.

- C'est ce qu'on appelle l'héritage prototypique et diffère de l'héritage de classe (Java par exemple).
- Cela permet de partager des fonctionnalités entre plusieurs objets sans dupliquer le code.
- Pour trouver le prototype d'un objet :
  - Object.getPrototypeOf(x);

1. Création d'un objet avec un prototype
   - Pour comprendre les prototypes, commençons par créer un objet simple :

```
let person = {
    firstName: "John",
    lastName: "Doe",
    getFullName: function() {
        return this.firstName + " " + this.lastName;
    }
};
```

   - Pour créer un nouvel objet qui hérite des propriétés et méthodes de person, nous pouvons utiliser Object.create :

```
let employee = Object.create(person);
employee.jobTitle = "Developer";

console.log(employee.getFullName()); // "John Doe"
console.log(employee.jobTitle); // "Developer"
```

LES PROTOTYPES

Prototypes avec les Fonctions Constructeurs

Les fonctions constructrices sont une autre manière de créer des objets avec des prototypes

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

Person.prototype.getFullName = function() {
    return this.firstName + " " + this.lastName;
};

let john = new Person("John", "Doe");
console.log(john.getFullName()); // "John Doe"
```

Héritage avec les Prototypes

Pour créer une chaîne de prototypes (héritage), nous pouvons définir le prototype d'une fonction constructeur comme une instance d'une autre fonction constructeur :

```
function Employee(firstName, lastName, jobTitle) {
    Person.call(this, firstName, lastName);
    this.jobTitle = jobTitle;
}

Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;

let jane = new Employee("Jane", "Smith", "Designer");
console.log(jane.getFullName()); // "Jane Smith"
console.log(jane.jobTitle); // "Designer"
```

TESTER LE PROTOTYPE D'UN OBJET

TESTER UN PROTOTYPE

instanceof

La méthode instanceof permet de vérifier si un objet est une instance d'un constructeur particulier. Cela fonctionne en vérifiant la chaîne de prototypes.

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(john instanceof Person); // true
console.log(john instanceof Object); // true
```

isPrototypeOf

La méthode isPrototypeOf permet de vérifier si un objet est dans la chaîne de prototypes d'un autre objet.

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(Person.prototype.isPrototypeOf(john)); // true
console.log(Object.prototype.isPrototypeOf(john)); // true
```

TESTER UN PROTOTYPE

Object.getPrototypeOf

La méthode Object.getPrototypeOf permet d'obtenir le prototype d'un objet. Vous pouvez ensuite comparer ce prototype avec un autre objet.

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(Object.getPrototypeOf(john) === Person.prototype); // true
console.log(Object.getPrototypeOf(john) === Object.prototype); // false
```

__proto__

Bien que l'utilisation de __proto__ soit déconseillée en faveur de Object.getPrototypeOf, elle est toujours disponible pour obtenir le prototype d'un objet.

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(john.__proto__ === Person.prototype); // true
console.log(john.__proto__ === Object.prototype); // false
```

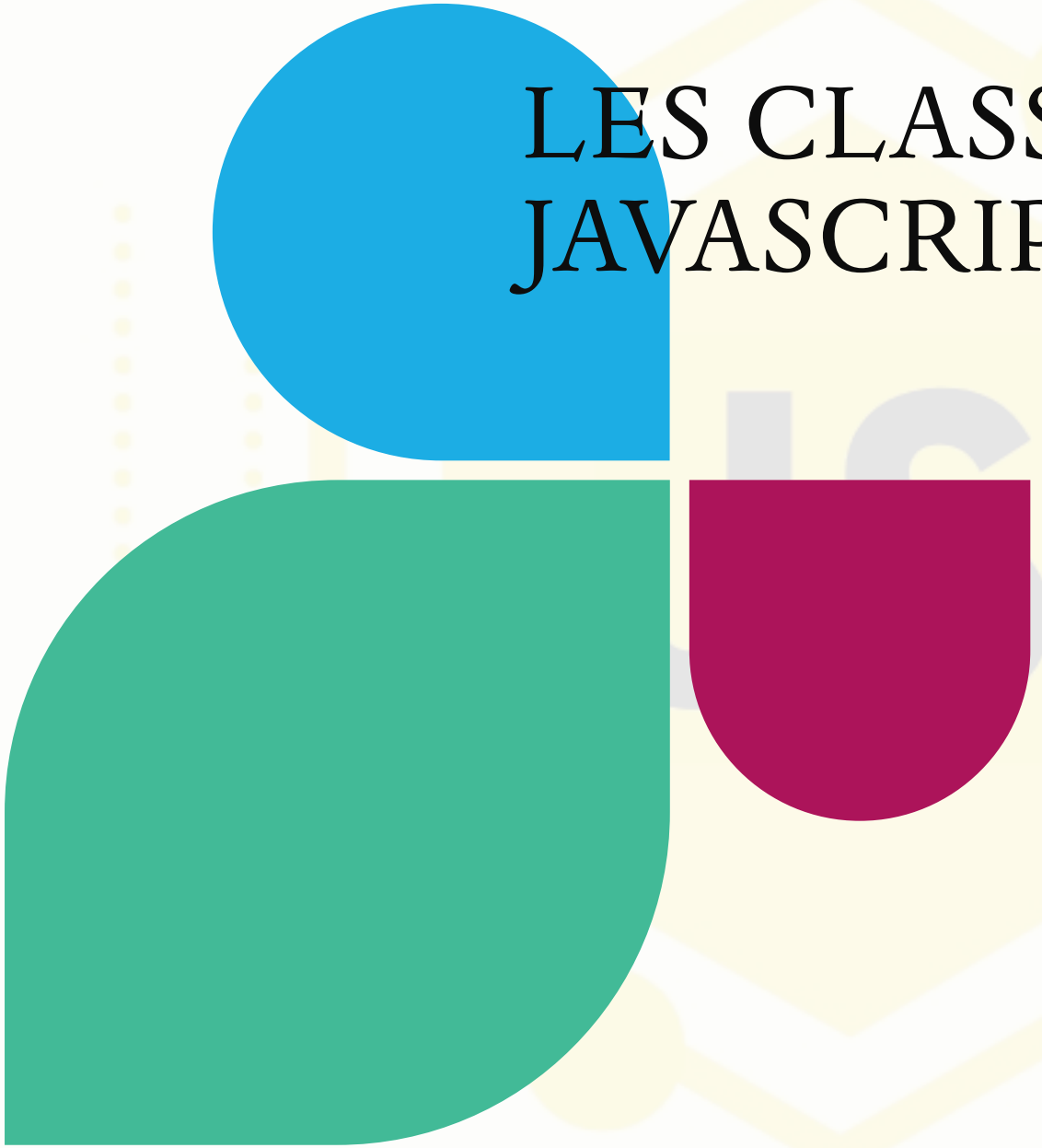
TESTER UN PROTOTYPE

Pour vérifier le type d'un objet, vous pouvez utiliser Object.prototype.toString.call.

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

let john = new Person("John", "Doe");

console.log(Object.prototype.toString.call(john)); // "[object Object]"
console.log(Object.prototype.toString.call(Person.prototype)); // "[object Object]"
```

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

let john = new Person("John", "Doe");

// Utiliser instanceof
console.log(john instanceof Person); // true
console.log(john instanceof Object); // true

// Utiliser isPrototypeOf
console.log(Person.prototype.isPrototypeOf(john)); // true
console.log(Object.prototype.isPrototypeOf(john)); // true

// Utiliser Object.getPrototypeOf
console.log(Object.getPrototypeOf(john) === Person.prototype); // true
console.log(Object.getPrototypeOf(john) === Object.prototype); // false

// Utiliser __proto__
console.log(john.__proto__ === Person.prototype); // true
console.log(john.__proto__ === Object.prototype); // false

// Utiliser Object.prototype.toString.call
console.log(Object.prototype.toString.call(john)); // "[object Object]"
console.log(Object.prototype.toString.call(Person.prototype)); // "[object Object]"
```

LES CLASSES EN JAVASCRIPT



CLASSE EN ES6

Avec ES6, JavaScript a introduit les classes, qui sont une syntaxe plus simple pour travailler avec les prototypes :

- Une classe en JavaScript est définie à l'aide du mot-clé class.
- Pour créer une instance de la classe Person, utilisez le mot-clé new
- Les classes en JavaScript supportent l'héritage via le mot-clé extends.

```
class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName() {
        return this.firstName + " " + this.lastName;
    }
}

class Employee extends Person {
    constructor(firstName, lastName, jobTitle) {
        super(firstName, lastName);
        this.jobTitle = jobTitle;
    }
}

let jim = new Employee("Jim", "Brown", "Manager");
console.log(jim.getFullName()); // "Jim Brown"
console.log(jim.jobTitle); // "Manager"
```

UTILISATION DE _ ET # DANS LES CLASSES

_ ET

Dans vos diverses recherches sur Javascript et l'objet, il se peut que vous trouviez du code avec _ ou # devant les attributs.

Ils sont chacun une signification différente que vous allons détailler :

- # permet de désigner une variable comme private. Ce qui la rends inaccessible en dehors de la classe. Ceci correspond donc à l'encapsulation.
  - Pour avoir accès à l'attribut, il faudra passer par un getter et un setter.
- _ est une convention plus ancienne et moins stricte. Elle peut être utilisée dans des contextes où tu souhaites indiquer que certains attributs ne devraient pas être directement accessibles ou modifiés, mais où tu n'as pas besoin d'une encapsulation stricte.
  - Cependant, avec l'introduction des champs privés, il est généralement préférable d'utiliser # pour une meilleure encapsulation.
  - La variable peut être accessible en dehors de la classe.

CLASSE EN ES6

Méthodes Statiques

Les méthodes statiques sont définies avec le mot-clé static et peuvent être appelées directement sur la classe sans créer d'instance.

```
class MathUtils {
    static add(a, b) {
        return a + b;
    }
}

console.log(MathUtils.add(2, 3)); // 5
```

Getters et Setters

Les classes en JavaScript supportent également les getters et setters pour accéder et modifier les propriétés :

```
class Rectangle {
    constructor(width, height) {
        this._width = width;
        this._height = height;
    }

    get area() {
        return this._width * this._height;
    }

    set width(value) {
        this._width = value;
    }

    set height(value) {
        this._height = value;
    }
}

let rect = new Rectangle(10, 20);
console.log(rect.area); // 200
rect.width = 15;
console.log(rect.area); // 300
```

EXPORTER UNE CLASSE

Supposons que vous avez une classe Person dans un fichier nommé Person.js

```
// Person.js
export class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName() {
        return this.firstName + " " + this.lastName;
    }
}
```

Vous pouvez ensuite importer cette classe dans un autre fichier, par exemple main.js :

```
// main.js
import { Person } from './Person.js';

let john = new Person("John", "Doe");
console.log(john.getFullName()); // "John Doe"
```

- Utiliser des exports par défaut : Si vous souhaitez exporter une seule classe ou fonction par défaut, vous pouvez utiliser export default et ensuite importer cette classe sans accolades :

```
// Person.js
export default class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName() {
        return this.firstNam
    }
}
```

```
// main.js
import Person from './Person.js';

let john = new Person("John", "Doe");
console.log(john.getFullName()); // "John Doe"
```

```
<script type="module" src="./script/script.js"></script>
```

EN RÉSUMÉ

EN RÉSUMÉ

Objet simple avec prototype

```
person = {
    firstName: "John",
    lastName: "Doe",
    getFullName: function() {
        return this.firstName + " " + this.lastName;
    }
}
```

Utilisation de Object.create

```
employee = Object.create(person);
employee.jobTitle = "Developer";

employee
|
|-- __proto__ ----> person
|                   |
|                   |-- firstName: "John"
|                   |-- lastName: "Doe"
|                   |-- getFullName: function() { ... }
|
|-- jobTitle: "Developer"
```

EN RÉSUMÉ

Fonctions Constructeurs et Prototypes

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
}

Person.prototype.getFullName = function() {
    return this.firstName + " " + this.lastName;
};

let john = new Person("John", "Doe");

john
|
|-- __proto__ ----> Person.prototype
|                     |
|                     |-- getFullName: function() { ... }
|
|-- firstName: "John"
|-- lastName: "Doe"
```

Héritage avec les Prototypes

```
function Employee(firstName, lastName, jobTitle) {
    Person.call(this, firstName, lastName);
    this.jobTitle = jobTitle;
}

Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;

let jane = new Employee("Jane", "Smith", "Designer");

jane
|
|-- __proto__ ----> Employee.prototype
|                     |
|                     |-- __proto__ ----> Person.prototype
|                     |                     |
|                     |                     |-- getFullName: function() { ... }
|                     |
|                     |-- constructor: Employee
|
|-- firstName: "Jane"
|-- lastName: "Smith"
|-- jobTitle: "Designer"
```

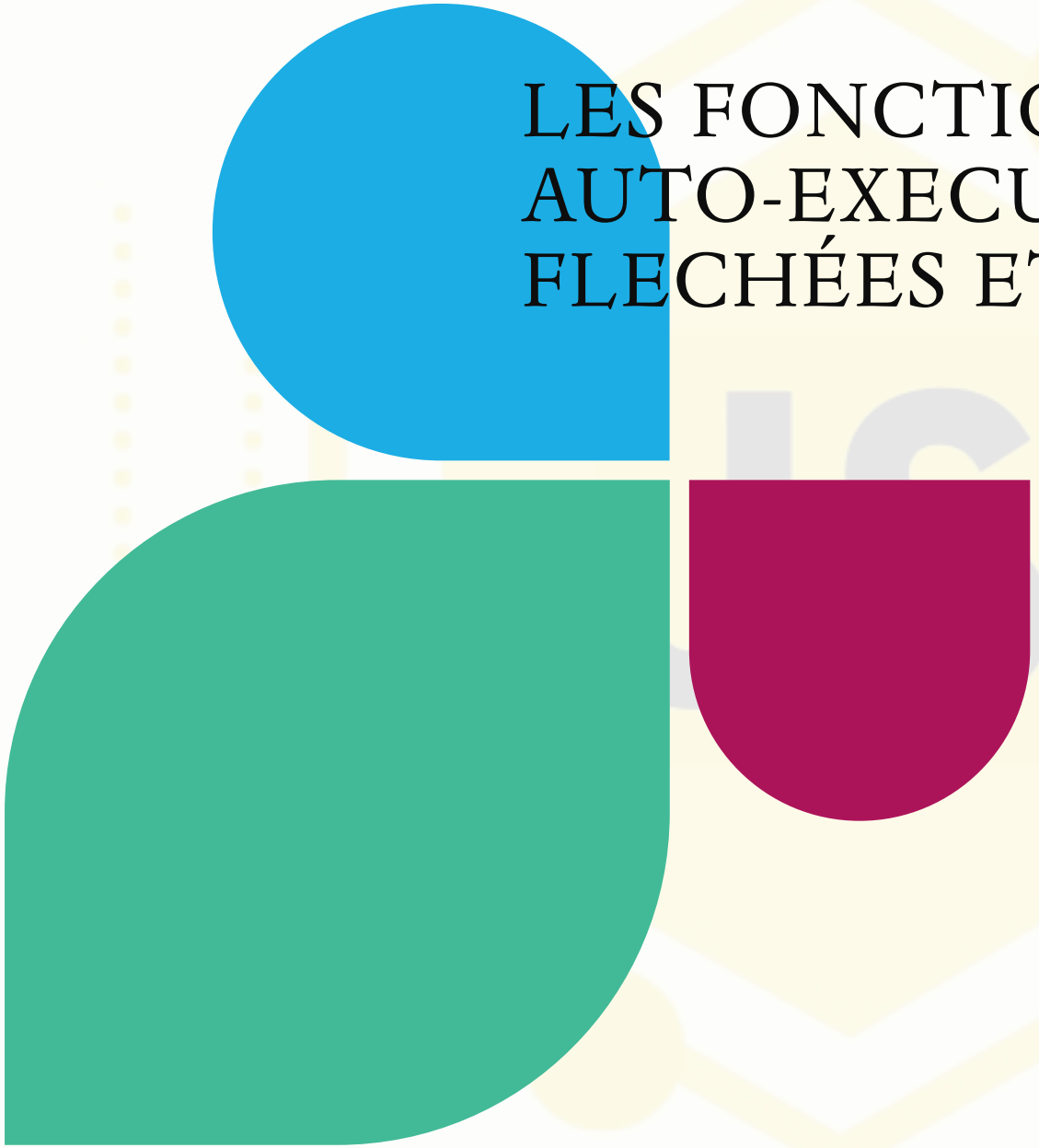
CLASSE EN ES6

L'objet jim hérite de Employee.prototype, qui hérite de Person.prototype.

Ces schémas montrent comment les objets héritent des propriétés et des méthodes via la chaîne de prototypes (__proto__).

```
class Person {
    constructor(firstName, lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    getFullName() {
        return this.firstName + " " + this.lastName;
    }
}

class Employee extends Person {
    constructor(firstName, lastName, jobTitle) {
        super(firstName, lastName);
        this.jobTitle = jobTitle;
    }
}

let jim = new Employee("Jim", "Brown", "Manager");

jim
|
|-- __proto__ ----> Employee.prototype
|                   |
|                   |-- __proto__ ----> Person.prototype
|                   |                   |
|                   |                   |-- getFullName: function() { ... }
|                   |
|                   |-- constructor: Employee
|
|-- firstName: "Jim"
|-- lastName: "Brown"
|-- jobTitle: "Manager"
```

LES FONCTIONS CALLBACK, AUTO-EXECUTABLES, FLECHÉES ET IMBRIQUÉES



CALLBACKS

Un callback est une fonction qui est passée comme argument à une autre fonction et qui est exécutée après qu'une certaine opération a été complétée.

- Les callbacks sont couramment utilisés dans les opérations asynchrones, comme les requêtes réseau ou les opérations de fichiers.

Dans cet exemple simple , sayGoodbye est une fonction de callback qui est exécutée après que greet a terminé son exécution.

```
function greet(name, callback) {
    console.log('Hello ' + name);
    callback();
}

function sayGoodbye() {
    console.log('Goodbye!');
}

greet('Alice', sayGoodbye);
```

AUTO-EXÉCUTABLES

Une fonction auto-exécutable (IIFE, Immediately Invoked Function Expression) est une fonction qui est définie et exécutée immédiatement après sa définition.

- Les IIFE sont souvent utilisées pour créer une portée privée et éviter la pollution de l'espace global.

La syntaxe impose simplement de faire suivre la définition de la fonction d'une paire de parenthèses afin de provoquer son exécution.

- Pour provoquer l'exécution immédiate d'une fonction, on peut encore l'englober dans une autre paire de parenthèses sans oublier de la faire suivre par sa paire de parenthèses

```
var test = function() {
    console.log('hello world');
}();

(function() {
    console.log('hello world');
}());

(function() {
    console.log('hello world');
})();
```

FONCTIONS FLÉCHÉES

- Syntaxe très compacte, rapide à écrire, utilisant le signe =>

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Functions/Arrow_functions

```
([param] [, param]) => {
  /* instructions */
}

(param1, param2, paramx) => expression;

// équivalent à
(param1, param2, paramx) => {
  return expression;
}

  // Parenthèses non nécessaires quand il n'y a qu'un seul argument
param => expression;

// Une fonction sans paramètre peut s'écrire avec un couple
// de parenthèses
() => {
  /* instructions */
}

(param1 = valeurDefaut1, param2, paramN = valeurDefautN) => {
  /* instructions */
}
```

FONCTIONS IMBRIQUÉES

- Une fonction peut avoir une ou plusieurs fonctions internes.
- Ces fonctions imbriquées sont dans la portée de la fonction externe.
- La fonction interne peut accéder aux variables et aux paramètres de la fonction externe. Cependant, la fonction externe ne peut pas accéder aux variables définies dans les fonctions internes.

```
function afficheMessage(prenom)
{
    function disBonjour() {
        alert("Bonjour " + prenom);
    }
    return disBonjour();
}

afficheMessage("Michel");     // Affiche Bonjour Michel
```

FONCTIONS IMBRIQUÉES AVEC CLOSURES

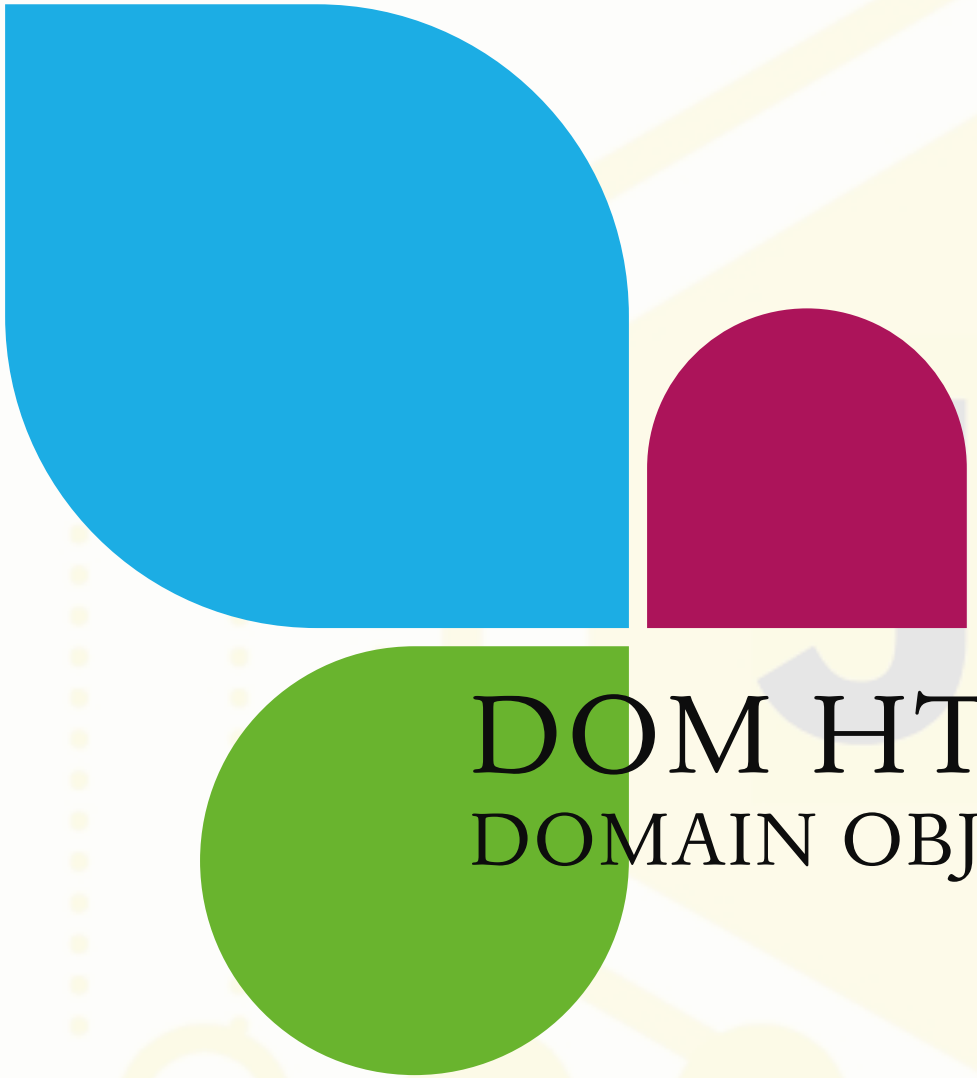
Les closures permettent à une fonction imbriquée d'accéder aux variables de la portée englobante, même après que la fonction externe a terminé son exécution.

```javascript
// Définition de la fonction externe
function outerFunction(outerVariable) {
    // Définition de la fonction interne
    return function innerFunction(innerVariable) {
        // Affichage des variables
        console.log('Outer Variable:', outerVariable);
        console.log('Inner Variable:', innerVariable);
    };
}

// Appel de la fonction externe avec 'outside'
const newFunction = outerFunction('outside');
// newFunction est maintenant une référence à innerFunction
// avec outerVariable fixé à 'outside'

// Appel de la fonction interne avec 'inside'
newFunction('inside');
// innerFunction affiche 'Outer Variable: outside'
// et 'Inner Variable: inside'
```

- outerFunction retourne une fonction imbriquée (innerFunction) qui capture la variable outerVariable grâce à la closure.
- newFunction est une référence à innerFunction avec outerVariable fixé à 'outside'.
- Lorsque vous appelez newFunction('inside'), innerFunction affiche les deux variables : outerVariable (qui est 'outside') et innerVariable (qui est 'inside').

```javascript
// Définition de la fonction createCounter
function createCounter() {
    // Initialisation de la variable count à 0
    let count = 0;

    // Retourne une fonction anonyme
    return function() {
        // Incrémente count de 1
        count++;
        // Retourne la nouvelle valeur de count
        return count;
    };
}

// Appel de createCounter et assignation de la fonction retournée à counter
const counter = createCounter();

// Appel de la fonction retournée (counter)
console.log(counter()); // 1 (count est incrémenté de 0 à 1)
console.log(counter()); // 2 (count est incrémenté de 1 à 2)
console.log(counter()); // 3 (count est incrémenté de 2 à 3)
```

La fonction anonyme capture la variable count grâce à la closure, ce qui permet de maintenir l'état de count entre les appels.



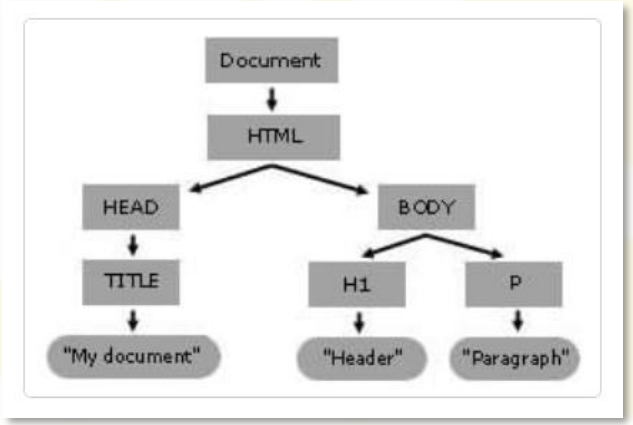
DOM HTML
DOMAIN OBJECT MODEL

DÉFINITION

Le DOM ou Document Object Model est une interface de programmation (ou API) pour les documents HTML

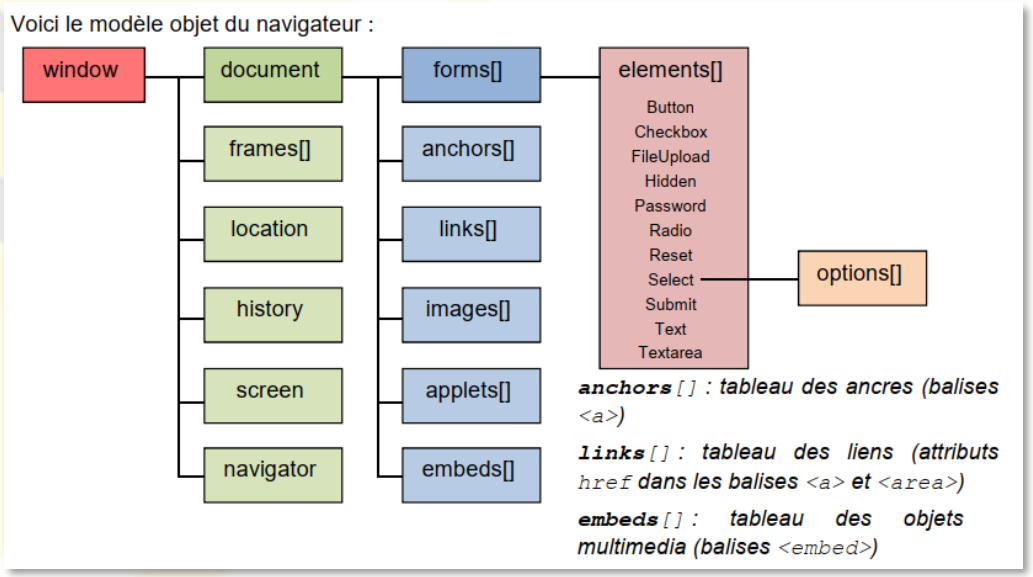
C'est donc un ensemble d'outils qui permettent de faire communiquer entre eux, dans le cas présent, les langages HTML et JavaScript.

- Standard établit par W3C



A la racine, l'objet window représente l'instance du navigateur :

- l'objet location qui symbolise la barre d'adresse,
- l'objet history qui représente l'historique des pages visitées par l'utilisateur
- l'objet document qui représente la page Web en cours, le contenu du <body> Html, lui-même référençant tous ses éléments.



L'OBJET WINDOW

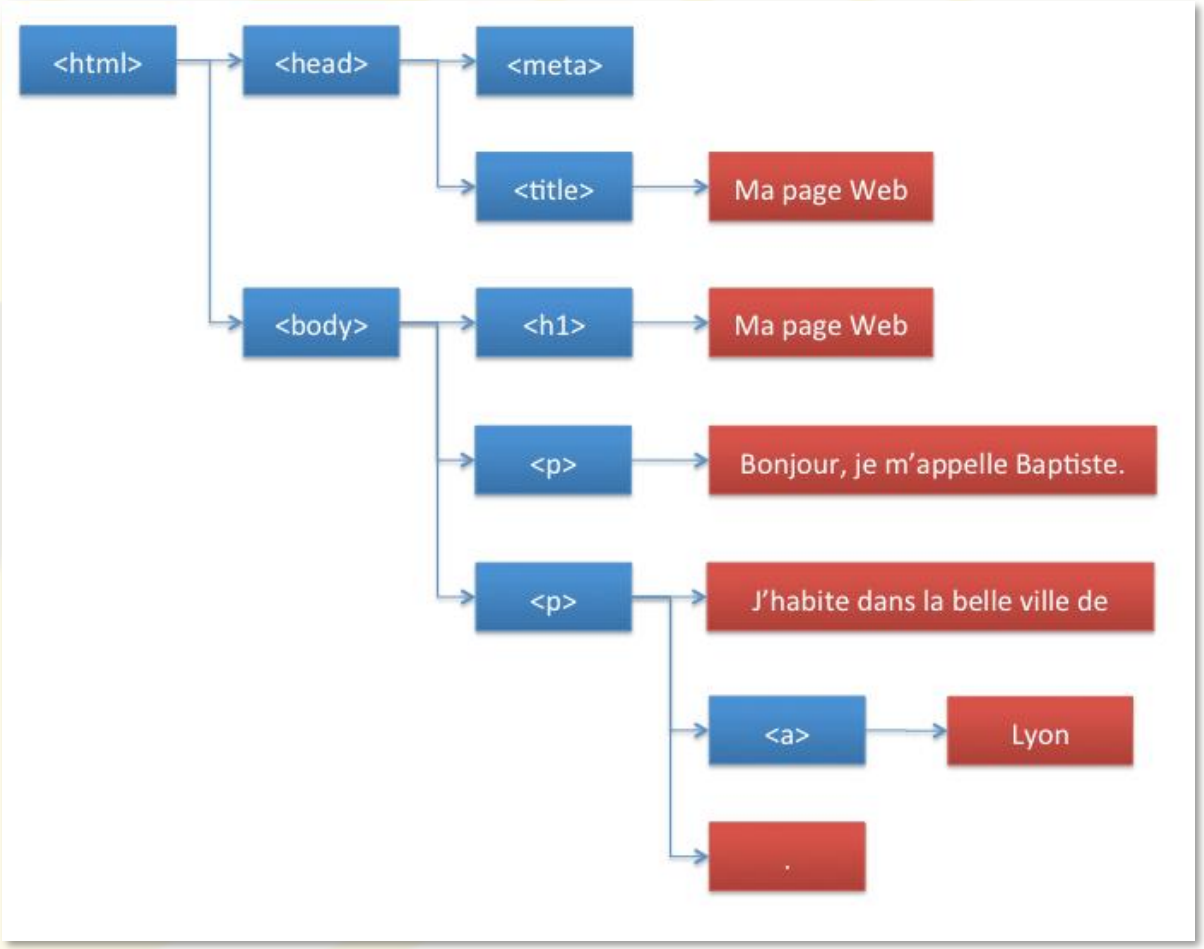
l'objet window qui représente une fenêtre contenant un document DOM

```
> console.log(window.location);
                                                                    VM162:1
  Location {ancestorOrigins: DOMStringList, href: 'http://tpform/t
  emplate.html?lastName=b&firstName=b…DateStart=&periodDateEnd=&nu
  mberDay=&code01=01_01', origin: 'http://tpform', protocol: 'htt
  p:', host: 'tpform', …}
< undefined
> console.log(window.location.search);
  ?                                                                 VM284:1
  lastName=b&firstName=b&study=CDA&option=optionOne&awayDate=2022-
  03-
  07&awayTimeStart=10&awayTimeEnd=11&periodDateStart=&periodDateEnd=
  &numberDay=&code01=01_01
< undefined
> console.log(window.location.search.substring(1));
  lastName=b&firstName=b&study=CDA&option=optionOne&awayDate VM367:1
  =2022-03-
  07&awayTimeStart=10&awayTimeEnd=11&periodDateStart=&periodDateEnd=
  &numberDay=&code01=01_01
```

TYPE DE NŒUD (NODE)

Dans le DOM, 2 types de nœuds :

- Ceux en bleu qui correspondent à des éléments HTML comme body, p. Ces nœuds peuvent avoir des sous-nœuds, appelés fils ou enfants (children) (nodeType = ELEMENT_NODE)
- Ceux en rouge, qui correspondent au contenu textuel de la page. Ces nœuds ne peuvent avoir de fils. (nodeType = TEXT_NODE)
- Document étant l'élément <html>



PARCOURIR LES NŒUDS

```html
<body>
    <h1>Les sept merveilles du monde</h1>
    <p>Connaissez-vous les merveilles du monde ?</p>
    <div id="contenu">
        <h2>Merveilles du monde antique</h2>
        <p>Cette liste nous vient de l'Antiquité.</p>
        <ul class="merveilles" id="antiques">
            <li class="existe">La pyramide de Khéops</li>
            <li>Les jardins suspendus de Babylone</li>
            <li>La statue de Zeus</li>
            <li>Le temple d'Artémis</li>
            <li>Le mausolée d'Halicarnasse</li>
            <li>Le Colosse de Rhodes</li>
            <li>Le phare d'Alexandrie</li>
        </ul>
        <h2>Nouvelles merveilles du monde</h2>
        <p>Cette liste a été établie en 2009 à la suite d'un vote par Internet.
        </p>
        <ul class="merveilles" id="nouvelles">
            <li class="existe">La Grande Muraille de Chine</li>
            <li class="existe">Pétra</li>
            <Li class="existe">Le Christ du Corcovado</Li>
            <Li class="existe">Machu Picchu</Li>
            <li class="existe">Chichén Itzá</li>
            <li class="existe">Le Colisée</li>
            <li class="existe">Le Taj Mahal</li>
        </ul>
        <h2>Références</h2>
        <ul>
            <li><a href="https://fr.wikipedia.org/wiki/
            Sept_merveilles_du_monde">Merveilles antiques</a></li>
            <li><a href="https://fr.wikipedia.org/wiki/
            Sept_nouvelles_merveilles_du_monde">Nouvelles merveilles</a></li>
        </ul>
    </div>
```

```javascript
// etape 1
var h = document.head; // La variable h contient l'objet head du DOM
console.log(h);
// etape 2
var b = document.body; // La variable b contient l'objet body du DOM
console.log(b);
// etape 3
// type de noeud
if (document.body.nodeType === document.ELEMENT_NODE) {
    console.log("Body est un noeud élément");
} else {
    console.log("Body est un noeud textuel");
}
// etape 4
// Accéder aux enfants d'un nœud élément
// Accès au premier enfant du noeud body ??
console.log(document.body.childNodes[0]);
/*
les espaces entre les balises ainsi que les retours à la ligne dans le code
HTML sont considérés par le navigateur comme des nœuds textuels. Ici, le noeud
h1 n'est donc que le deuxième enfant du nœud body.
*/
console.log(document.body.childNodes[1]);
// parcours des noeuds enfants
// Affiche les noeuds enfant du noeud body
for (let i = 0; i < document.body.childNodes.length; i++) {
    console.log(document.body.childNodes[i]);
}
// Accéder au parent d'un nœud
var h1 = document.body.childNodes[1];
console.log(h1.parentNode); // Affiche le noeud body

console.log(document.parentNode); // Affiche null : document n'a aucun noeud
parent
```

ACCÉDER AU DOCUMENT

Chaque élément du DOM est un objet Javascript avec ses propriétés et ses fonctions pour le manipuler.

Tout commence avec le document qui représente la page entière.

```
let element = document.getElementById(id);

let elements = document.getElementsByClassName(names); // ou:
elements = rootElement.getElementsByClassName(names);

elements = document.getElementsByTagName(nom);

elements = document.getElementsByName(nom)

let el = document.querySelector(".maclasse");

let matches = myBox.querySelectorAll("p");
```

document.getElementById()

- Retrouver un élément précis par son id

document.getElementsByClassName()

- Retrouver tous les éléments par la classe

document.getElementsByTagName()

- Retrouver tous les éléments avec un nom de balise

document.getElementsByName()

- Retrouver tous les éléments portant un nom donné dans le document HTML

document.querySelector()

- Retrouver le 1er élément correspondant au sélecteur CSS ou groupe de sélecteurs CSS

document.querySelectorAll()

- Retrouver tous les éléments correspondant au sélecteur CSS ou groupe de sélecteurs CSS

RECHERCHE DEPUIS UN ÉLÉMENT

Il n'y a pas qu'avec document que vous pouvez rechercher des éléments.

Comme nous l'avons vu, chaque élément est un objet JavaScript avec ses propriétés et ses fonctions.

Et parmi ces dernières, il en existe pour parcourir les enfants et le parent de chaque élément !

element.children

- Retourne une liste d'enfant de l'élément

element.parentElement

- Retourne l'élément parent

element.nextElementSibling et element.previousElementSibling

- Permet de naviguer vers l'élément suivant / précédent de même niveau

```
<div id="parent">
    <div id="previous">Précédent</div>
    <div id="main">
        <p>Paragraphe 1</p>
        <p>Paragraphe 2</p>
    </div>
    <div id="next">Suivant</div>
</div>
```

```
const elt = document.getElementById('main');
```

elt.children = les éléments de type p enfant de #main
elt.parentElement = div qui à l'id parent
elt.nextElementSibling = next
elt.previousElementSibling = previous

MODIFIER LE DOM

Deux propriétés principales :

- InnerHTML
  - Récupère ou définit la contenu HTML d'un élément du DOM
- textContent
  - Récupère son contenu textuel sans le balisage HTML du DOM

- Les attributs et les classes :
  - Modifier des classes d'un élément :
    - Possible d'accéder à la liste des classes d'un élément avec la propriété classList
      - add(string) : ajoute une classe
      - remove(string) : supprime la classe
      - contains(string) : vérifie si la classe existe
      - replace(old, new) : remplace l'ancienne classe par la nouvelle
  - Changer les styles d'un élément :
    - Possible d'accéder au style avec la propriété style
      - element.style.backgroundColor = '#000';
  - Modifier les attributs d'un élément :
    - Définir ou remplacer les attributs avec la fonction setAttribute, getAttribute, removeAttribute

MODIFIER LE DOM

On peut également :

1. Créer de nouveaux éléments.
2. Ajouter des enfants.
3. Supprimer et remplacer des éléments.

- La fonction createElement permet de créer de nouveaux éléments.
- Plusieurs façons d'ajouter un élément :
  - appendChild
- Les fonctions removeChild et replaceChild permettent de supprimer ou remplacer un élément

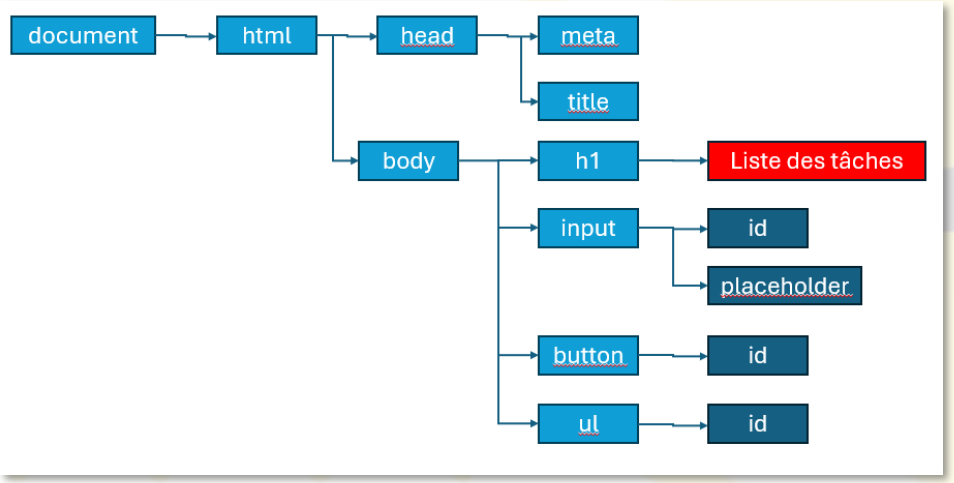
```
const newElt = document.createElement("div");
let elt = document.getElementById("main");
elt.appendChild(newElt);

elt.removeChild(newElt);    // Supprime l'élément newElt de l'élément elt
elt.replaceChild(document.createElement("article"), newElt);    // Remplace l'élément newElt
par un nouvel élément de type article
```

```
const newElt = document.createElement("div");
let elt = document.getElementById("main");

elt.appendChild(newElt);
```

```
const newElt = document.createElement("div");
```

EXEMPLE

Ci contre, un fichier HTML pour lister des tâches que l'utilisateur saisie.



dom.html > ...

```html
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Liste de Tâches</title>
    <!-- css -->
    <link rel="stylesheet" href="./CSS/swapimeteo.css">
</head>
<body>
    <h1>Liste de Tâches</h1>
    <input type="text" id="nouvelleTache" placeholder="Ajouter une tâche">
    <button id="ajouter">Ajouter</button>
    <ul id="listeTaches"></ul>

    <script type="text/javascript" src="./scriptJS/dom.js"></script>
</body>
</html>
```

EXEMPLE

Le script associé au fichier précédemment va construire DOM en fonction de la saisie utilisateur en créant les nœuds li et les ajouter au nœud ul


```javascript
// script JS de manipulation du DOM
function ajouterTache() {

    // recuperation de la valeur de l'element dont l'id est nouvelleTache dans le DOM
    // utilisation de trim pour enlever les espaces vide
    let tache = document.getElementById("nouvelleTache").value.trim() ;
    // test si aucune saisie
    if (tache !== "") {
        // récuperation de l'element dont l'id est listeTaches dans le DOM
        let liste = document.getElementById("listeTaches");
        // création d'un element li
        let element = document.createElement("li");
        // ajout du texte du li
        element.textContent = tache;

        // ajout d'un evenement sur l'element li
        element.onclick = function () {
            // suppression de l'element
            liste.removeChild(element);
        };

        // ajout de l'element li au parent
        liste.appendChild(element);
        // reset du champs input
        document.getElementById("nouvelleTache").value = "";
    } else {
        // affichage d'une alerte de saisie
        alert("Merci de saisir une tâche !!");
    }
}

// au chargement de la page HTML
document.addEventListener('DOMContentLoaded', function () {
    // ajout d'un evenement sur le bouton sur click
    document.getElementById("ajouter").addEventListener('click', ajouterTache);
});
```

LES ÉVÈNEMENTS

ACTIONS !!

ÉCOUTER LES ÉVÈNEMENTS

- Réaction à une action émise par l'utilisateur comme le clic, la saisie etc...
- Un événement en JavaScript est représenté par un nom (click , mousemove ...) et une fonction que l'on nomme un callback
- Par défaut, un événement est propagé, transmis à l'élément parent jusqu'à la racine du document tant qu'on ne l'a pas traité.
- La fonction de callback est appelée à chaque fois que l'action est désirée

AddEventListener permet d'écouter tous type d'évenements

- PreventDefault() est une option permettant de supprimer le comportement par défaut de l'événement
- stopPropagation() permet d'empêcher la propagation de l'événement au parent

```
const elt = document.getElementById('mon-lien');    // On récupère l'élément sur lequel on veut détecter le clic
elt.addEventListener('click', function() {           // On écoute l'événement click
    elt.innerHTML = "C'est cliqué !";                // On change le contenu de notre élément pour afficher "C'est cliqué !"
});
```

```
const elt = document.getElementById('mon-lien');    // On récupère l'élément sur lequel on veut détecter le clic
elt.addEventListener('click', function(event) {      // On écoute l'événement click, notre callback prend un paramètre que nous avons appelé event ici
    event.preventDefault();                          // On utilise la fonction preventDefault de notre objet event pour empêcher le comportement par défaut de cet élément lors du clic de la souris
});
```

LA PROPAGATION DES ÉVÉNEMENTS

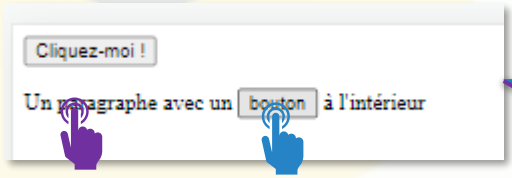
Le DOM représente une page web sous la forme d'une hiérarchie de nœuds.

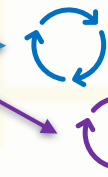
Les événements déclenchés sur un nœud enfant vont se déclencher ensuite sur son nœud parent, puis sur le parent de celui-ci, et ce jusqu'à la racine du DOM (la variable document).

C'est ce qu'on appelle la propagation des événements.

```html
<body>
    <button id="bouton">Cliquez-moi !</button>

    <p id="para">Un paragraphe avec un
        <button id="propa">bouton</button> à l'intérieur
    </p>


    <script src="ressources/event.js"></script>
</body>
```

```js
// Gestion du clic sur le document
document.addEventListener("click", function () {
    console.log("Gestionnaire document");
});
// Gestion du clic sur le paragraphe
document.getElementById("para").addEventListener("click", function () {
    console.log("Gestionnaire paragraphe");
});
// Gestion du clic sur le bouton
document.getElementById("propa").addEventListener("click", function (e) {
    console.log("Gestionnaire bouton");
});
```





```
Gestionnaire bouton                    event.js:86
Gestionnaire paragraphe                event.js:82
Gestionnaire document                  event.js:78
```

```
Gestionnaire paragraphe                event.js:82
Gestionnaire document                  event.js:78
```

LA PROPAGATION DES ÉVÉNEMENTS

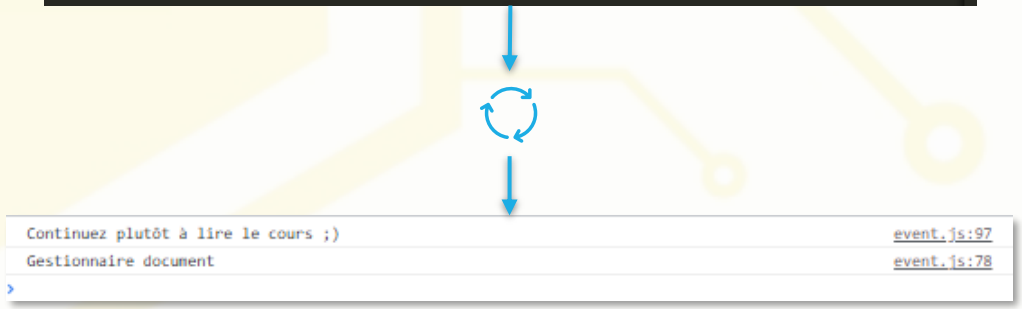
Stopper la propagation

```
// Gestion du clic sur le bouton
document.getElementById("propa").addEventListener("click", function (e) {
    console.log("Gestionnaire bouton");
    e.stopPropagation(); // Arrêt de la propagation de l'événement
});
```

Modifier le comportement par défaut

```
<p>Du temps à perdre ? <a id="interdit" href="https://9gag.com/">Clique
ici</a></p>
```

```
// Gestion du clic sur le lien interdit
document.getElementById("interdit").addEventListener("click", function (e)
    console.log("Continuez plutôt à lire le cours ;)");
    e.preventDefault(); // Annulation de la navigation vers la cible du lie
});
```



TYPE D'ÉVÉNEMENTS

Les événements que les éléments du DOM peuvent déclencher sont très nombreux.

Le tableau ci-contre présente les principales catégories d'événements.

Quel que soit le type d'événement, son déclenchement s'accompagne de la création d'un objet Event

Catégorie	Exemples
Événements clavier	Appui ou relâchement d'une touche du clavier
Événements souris	Clic avec les différents boutons, appui ou relâchement d'un bouton de la souris, survol d'une zone avec la souris
Événements fenêtre	Chargement ou fermeture de la page, redimensionnement, défilement *(scrolling)*
Événements formulaire	Changement de cible de saisie (focus), envoi d'un formulaire

```
// Ajout d'un gestionnaire qui affiche le type et la cible de l'événement
document.getElementById("bouton").addEventListener("click", function (e) {
    console.log("Evènement : " + e.type +
        ", texte de la cible : " + e.target.textContent);
});
```

keyup - keydow - keypress

clic - mousemove

submit - reset

load - beforeunload

LES ÉVÈNEMENTS

La souris

Avec mousemove, on peut détecter le mouvement de la souris.

Cet évènement fournit un objet MouseEvent qui permet de récupérer

- clientX / clientY : position de la souris
- offsetX / offsetY : position par rapport à l'élément
- pageX / pageY position par rapport au document
- screenX / screenY position par rapport à la fenêtre
- movementX / movementY position par rapport à la position lors du dernier évènement mousemove

Lire un champ texte

L'évènement change pour les éléments input, select, textarea, checkbox, radio est déclenché lorsque le champ perd le focus

L'évènement input permet de connaitre les modifications d'un élément en cours par l'utilisateur.

```
<label>Choose an ice cream flavor:
    <select id="ice-cream" name="ice-cream">
        <option value="">Select One …</option>
        <option value="chocolate">Chocolate</option>
        <option value="strawberry">Strawberry</option>
        <option value="vanilla">Vanilla</option>
    </select>
</label>
```

```
document.addEventListener('DOMContentLoaded',function() {
    document.querySelector('select[name="ice-cream"]').onchange=changeEventHandler;
},false);

function changeEventHandler(event) {
    // You can use “this” to refer to the selected element.
    if(!event.target.value) alert('Please Select One');
    else alert('You like ' + event.target.value + ' ice cream.');
}
```

EXEMPLE

```html
<!DOCTYPE html>
<html lang="fr">

    <head>

        <!-- ENCODAGE -->
        <meta charset="UTF-8">
        <!-- description du site -->
        <meta name="description" content="Cours HTML">
        <!-- prise en charge du contexte mobile -->
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <!-- titre de la page -->
        <title>Document</title>
        <link rel="stylesheet" type="text/css" href="ressources/style/style.
        css">

    <body>

        <h1>TEST JS</h1>

        <input type="button" value="Change ce document." onclick="change()">
        <h2>H2</h2>
        <p>Paragraphe</p>


        <script StyleSheet="text/javascript" src="/ressources/script/script.
        js"></script>

    </body>


</html>
```

On peut aussi appliquer l'événement directement dans le html

```javascript
function change() {

    // document.getElementsByTagName ("H2") renvoie un NodeList du <h2>
    // éléments dans le document, et le premier est le nombre 0:
    let header = document.getElementsByTagName("H2").item(0);
    // le firstChild de l'en-tête est un noeud texte::
    header.firstChild.data = "A dynamic document";
    // maintenant l'en-tête est "Un document dynamique".

    let para = document.getElementsByTagName("P").item(0);
    para.firstChild.data = "This is the first paragraph.";

    // crée un nouveau noeud texte pour le second paragraphe
    let newText = document.createTextNode("This is the second paragraph.");
    // crée un nouvel Element pour le second paragraphe
    let newElement = document.createElement("P");
    // pose le texte dans le paragraphe
    newElement.appendChild(newText);
    // et pose le paragraphe à la fin du document en l'ajoutant
    // au BODY (qui est le parent de para)
    para.parentNode.appendChild(newElement);
}
```

LES FORMULAIRES
MANIPULATION ET INTERACTION

LES FORMULAIRES

Zone de texte - input et textarea

- required : oblige la saisie
- value : valeur de la zone de texte
- gestion du focus :
  - focus : saisie en cours dans la zone de texte
  - blur : changement de cible provoque un blur sur l'ancienne zone de texte qui avait le focus

```
// Affichage d'un message contextuel pour la saisie du pseudo
pseudoElt.addEventListener("focus", function () {
    document.getElementById("aidePseudo").textContent = "Entrez votre pseud
});
// Suppression du message contextuel pour la saisie du pseudo
pseudoElt.addEventListener("blur", function (e) {
    document.getElementById("aidePseudo").textContent = "";
});
```

Les cases à cocher et boutons radios

- change indique lorsque l'utilisateur modifie son choix.
- Case à cocher :
  - Event dispose d'une propriété checked (cochée / décochée)
- Boutons radio :
  - Balise input de type radio avec l'attribut name et value

```
// Affichage de la demande de confirmation d'inscription
document.getElementById("confirmation").addEventListener("change", function (e)
{
    console.log("Demande de confirmation : " + e.target.checked);
});
```

LES FORMULAIRES

Liste déroulante

- balise select + option
- change est déclenché sur les modifications apportées à la liste.

Comme pour les boutons radio, la propriété e.target.value de l'événement change contient la valeur de l'attribut value de la balise option associé au nouveau choix, et non pas le texte affiché dans la liste déroulante.

Formulaire et DOM

La balise form est l'élément à cibler pour accéder au contenu du formulaire par l'attribut elements rassemblant les champs de saisie

- soumission du formulaire
  - input de type submit et input de type reset
  - C'est à partir de cette soumission que nous allons tester et contrôler la saisie de l'utilisateur et en cas d'erreur utiliser preventDefault

CONTRÔLE DE LA SAISIE

CONTRÔLER L'UTILISATEUR

SUR LES ÉLÉMENTS DU FORMULAIRE

Vous avez aussi la possibilité de mettre en place des patterns sur la balise input afin de mettre en place un contrôle sur la saisie à la validation.

L'attribut pattern peut être utilisé pour les champs de type text, tel, email, url, password, search.

Les REGEX vont nous aider à mettre en place des contrôles sur la saisie.

```
<p>
    <label for="mdp">Mot de passe</label> :
    <input type="password" name="mdp" id="mdp"
    pattern="(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}"
    title="doit contenir au moins 1 chiffre, 1 majuscule, 1 minuscule
    et au moins 8 caractères" required>
    <span id="aideMdp"></span>
</p>
```

SUR LES DIFFÉRENTES PHASES DU FORMULAIRE

Le contrôle de validité peut se faire de plusieurs manières, éventuellement combinables :

- Soit au fur et à mesure de la saisie d'une donnée
  - input
- Soit à la fin de la saisie d'une donnée
  - blur
- Soit au moment où l'utilisateur soumet le formulaire.
  - submit

```
// Vérification de la longueur du mot de passe saisi
document.getElementById("mdp").addEventListener("input", function (e) {
    var mdp = e.target.value; // Valeur saisie dans le champ mdp
    var longueurMdp = "faible";
    var couleurMsg = "red"; // Longueur faible => couleur rouge
    if (mdp.length >= 8) {
        longueurMdp = "suffisante";
        couleurMsg = "green"; // Longueur suffisante => couleur verte
    } else if (mdp.length >= 4) {
        longueurMdp = "moyenne";
        couleurMsg = "orange"; // Longueur moyenne => couleur orange
    }
    var aideMdpElt = document.getElementById("aideMdp");
    aideMdpElt.textContent = "Longueur : " + longueurMdp; // Texte de l'aide
    aideMdpElt.style.color = couleurMsg; // Couleur du texte de l'aide
});

// Contrôle du courriel en fin de saisie
document.getElementById("courriel").addEventListener("blur", function (e) {
    var validiteCourriel = "";
    if (e.target.value.indexOf("@") === -1) {
        // Le courriel saisi ne contient pas le caractère @
        validiteCourriel = "Adresse invalide";
    }
    document.getElementById("aideCourriel").textContent = validiteCourriel;
});
```

EXEMPLE AVEC JAVASCRIPT

On peut également écrire nos fonctions de contrôle à la validation afin d'être plus précis sur nos contrôles.

```javascript
// méthode sans REGEX
// Contrôle du courriel en fin de saisie
document.getElementById("courriel").addEventListener("blur", function (e) {
    var validiteCourriel = "";
    if (e.target.value.indexOf("@") === -1) {
        // Le courriel saisi ne contient pas le caractère @
        validiteCourriel = "Adresse invalide";
    }
    document.getElementById("aideCourriel").textContent = validiteCourriel;
});

// méthode avec REGEX
// Contrôle du courriel en fin de saisie
document.getElementById("courriel").addEventListener("blur", function (e) {
    // Correspond à une chaîne de la forme xxx@yyy.zzz
    var regexCourriel = /.+@.+\..+/;
    var validiteCourriel = "";
    if (!regexCourriel.test(e.target.value)) {
        validiteCourriel = "Adresse invalide";
    }
    document.getElementById("aideCourriel").textContent = validiteCourriel;
});
```

LE VOCABULAIRE DU WEB

QUELQUES DÉFINITIONS

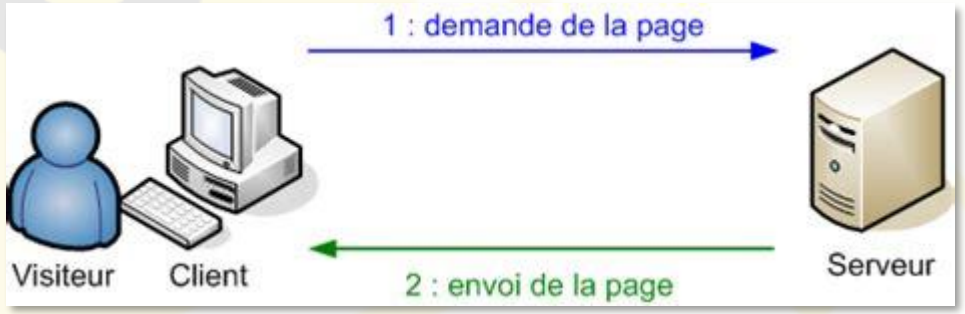
URL Uniform Resource Locator

- Format de nommage universel pour désigner une ressource sur Internet.
  - Protocole (HTTP, HTTPS, FTP…)
  - Nom de domaine ou l'adresse IP de l'ordinateur hébergeant la ressource.
  - Port logiciel utilisé par le protocole pour la mise en place de connexion (HTTP = 80 port par défaut)
  - La ressource demandée.

Anatomie de l'IP reçue par le navigateur

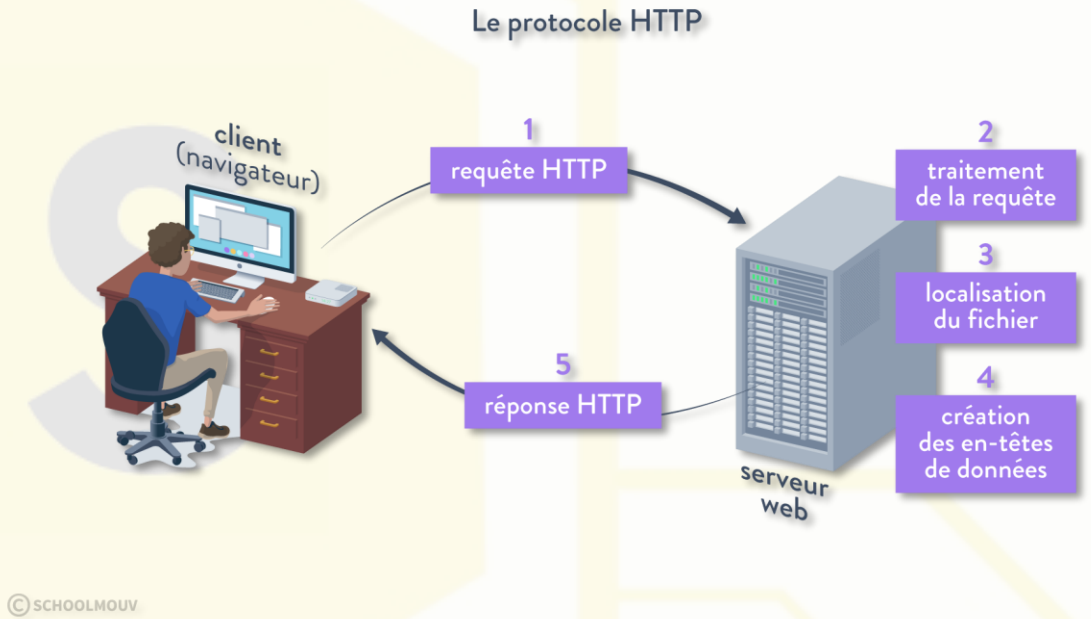
Requête HTTP HyperText Transfer Protocol

- Transfert de messages avec des en-têtes en codage MIME



PROTOCOLE HTTP

1. Le navigateur effectue une requête HTTP.
   - Une requête est composée :
     - D'un en-tête et du corps de la requête
   - Différentes méthodes :
     - GET, POST
     - HEAD, PUT, DELETE
2. Le serveur traite la requête puis envoie une réponse HTTP.
   - Une Réponse est composée :
     - D'un statut :
     - protocole utilisé
     - code de retour et de son libellé
   - D'un en-tête,
     - Du corps de la réponse



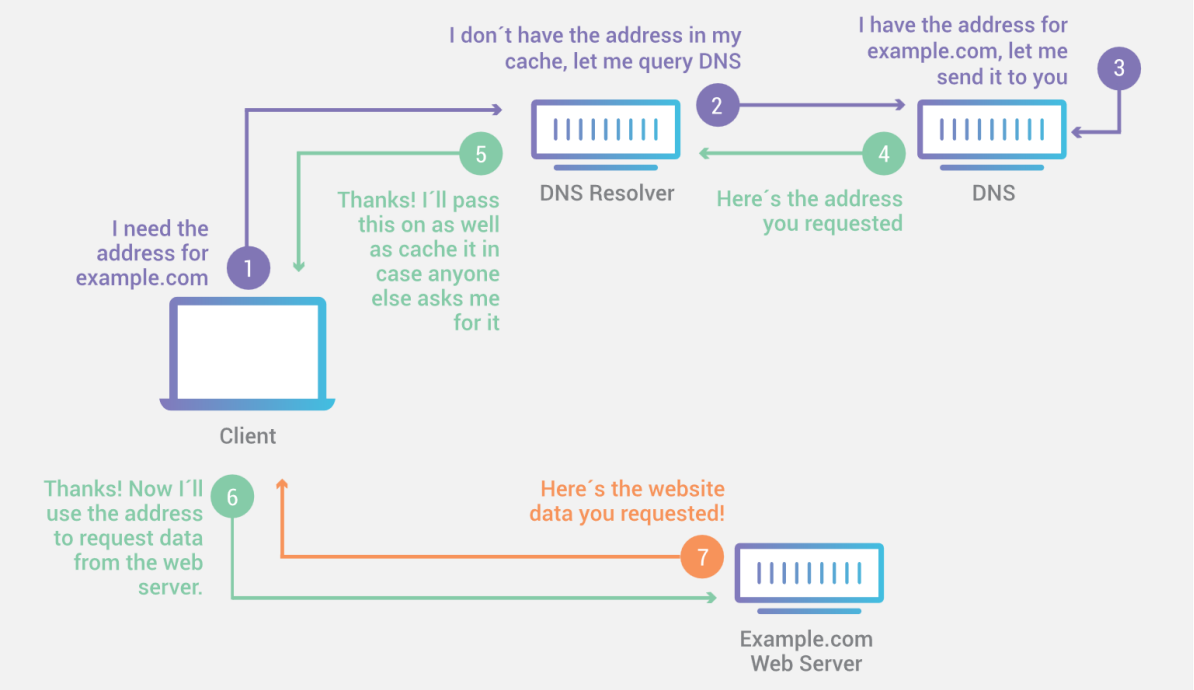
DOMAIN NAME SYSTEM

DNS : le DNS permet de traduire les noms de domaine en une adresse IP correspondant à une machine (serveur).

- Chaque appareil connecté à Internet dispose d'une adresse IP unique que les autres appareils utilisent afin de le trouver.

Grâce aux serveurs DNS, les internautes n'ont pas à mémoriser les adresses IP, ni les adresses IP alphanumériques plus récentes et plus complexes.

Ceci grâce aux fichiers de zone placés sur les serveurs DNS



REQUÊTES HTTP
COMMUNIQUER AVEC UN SERVEUR

QUELQUES NOTIONS

Quelques notions théoriques qu'il faut connaître avant de se lancer dans la communication avec un serveur en JavaScript.□□□□□□□

Service Web : programme s'exécutant sur un serveur accessible depuis internet et fournissant un service.

- Pour ce faire, il met à disposition une API.

API : Application Programming Interface - interface de communication avec les services Web, au travers de requêtes.

Requêtes : données qui respectent le protocole de communication et qui sont envoyées au serveur.

Protocoles : SMTP (envoi de mails), IMAP (réception de mails), HTTP et HTTPS, FTP, WebDAV, etc…

Requêtes HTTP asynchrones : AJAX (Asynchronous JavaScript and XML) est la technologie utilisée pour gérer ce type de requêtes.

- Cela permet de ne pas bloquer le navigateur pendant l'attente d'une réponse du serveur.

JSON : le format de données standard actuel pour l'échange des données sur le WEB.

LE PROTOCOLE HTTP(S)

HTTP : HyperText Transfert Protocol

- Communiquer avec un site internet, chargement des pages HTML, des styles CSS, etc…
- Envoie et récupération d'informations avec les formulaires.

- Méthodes :
  - GET : permet de récupérer des ressources
  - POST : permet de créer ou modifier une ressource
  - PUT : permet de modifier une ressource
  - DELETE : permet de supprimer une ressource
  - HEAD : demande des informations sur la ressource sans obtenir la ressource.
  - TRACE, OPTION, CONNECT…
- URL : l'adresse du service web à atteindre
- Données : les données qu'on envoie mais aussi qu'on reçoit
- Code HTTP : code numérique qui indique comment s'est déroulée la requête
  - 200 : tout s'est bien passé
  - 404 : la ressource n'existe pas
  - 500 : une erreur avec le service web
  - … https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

GET vs POST

GET

Cette méthode de requête existe depuis le début du Web. Elle est utilisée pour demander une ressource, par exemple un fichier HTML, au serveur Web.

- Paramètres URL

La requête GET peut recevoir des informations supplémentaires que le serveur Web doit traiter.

Ces paramètres d'URL sont simplement ajoutés à l'URL. La syntaxe est très simple :

- La chaîne de requête est introduite par un ? Et chaque paramètre est nommé, et composé donc d'un nom et d'une valeur : Nom=Valeur
- Si plusieurs paramètres doivent être inclus, ils sont reliés par un &
- un encodage des caractères spéciaux est efdectué (exemple : @).

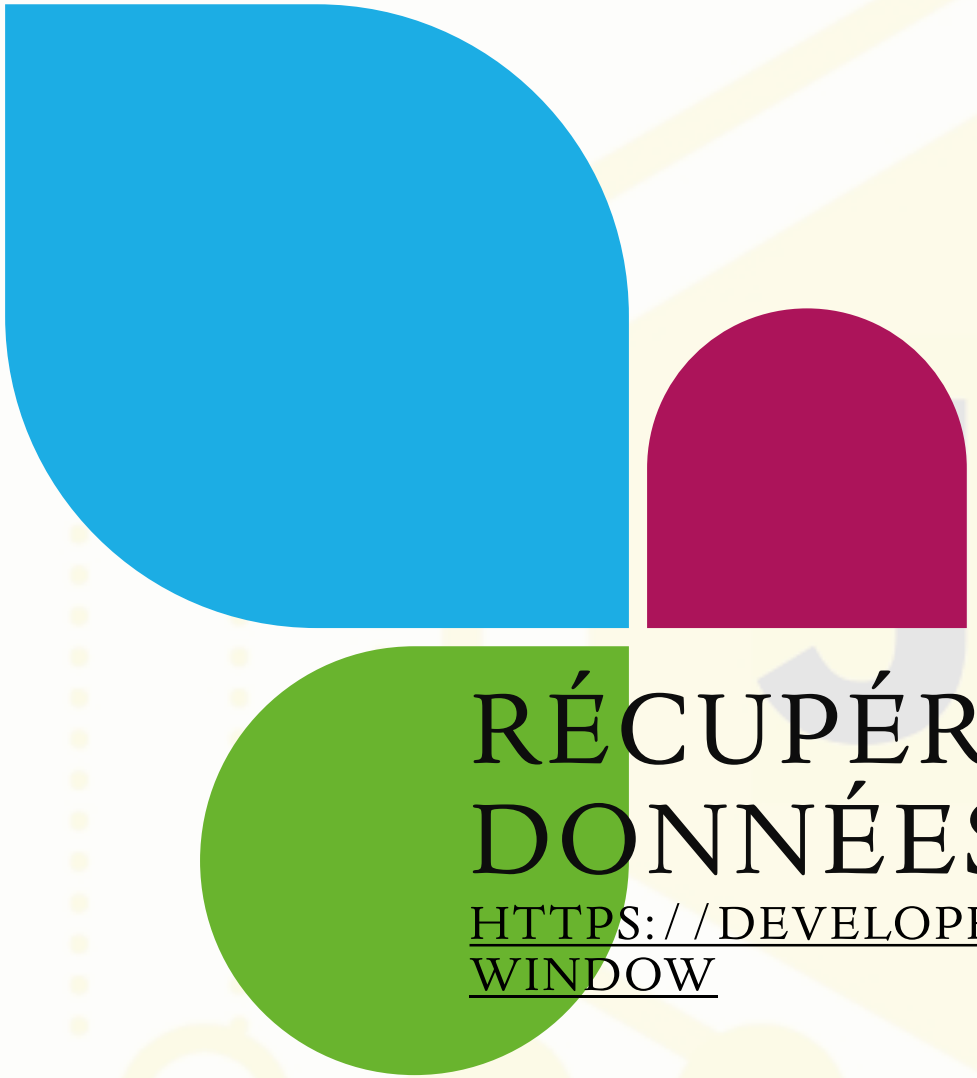
POST

Si vous souhaitez envoyer de grandes quantités de données, par exemple des images, ou des données confidentielles de formulaires au serveur, la méthode GET n'est pas idéale car toutes les données que vous envoyez sont écrites ouvertement dans la barre d'adresse du navigateur.

Dans ces cas, la méthode POST est la plus adaptée.

Cette méthode n'écrit pas les paramètres de l'URL, mais les ajoute à l'en-tête HTTP.

- Les requêtes POST sont principalement utilisées pour les formulaires en ligne.



RÉCUPÉRATION DES DONNÉES DU FORMULAIRE

HTTPS://DEVELOPER.MOZILLA.ORG/FR/DOCS/WEB/API/WINDOW

EXEMPLE

Soit le formulaire suivant :

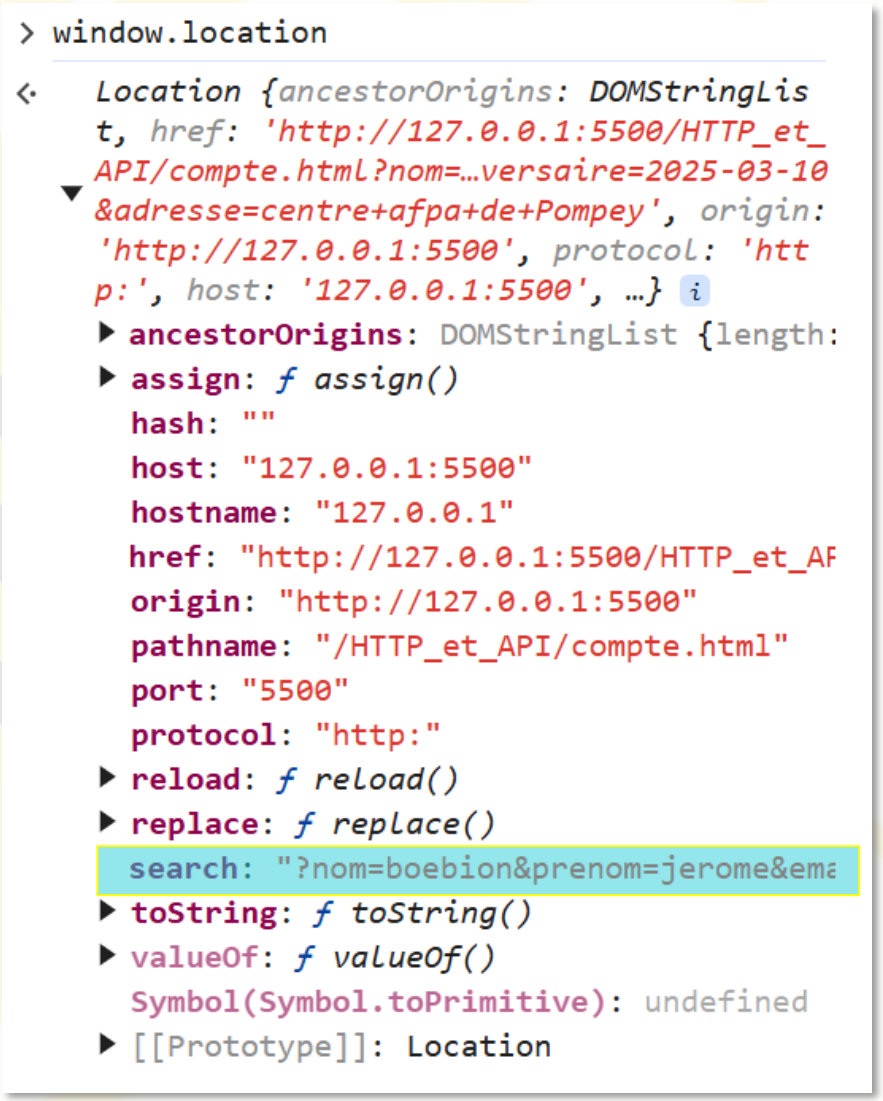

```html
<form id="userForm" action="compte.html" method="get">
    <div class="form-group">
        <label for="nom">Nom :</label>
        <input type="text" id="nom" name="nom" required pattern="[A-Za-zÀ-ÖØ-öø-ÿ\s]+">
        <div class="error" id="nomError"></div>
    </div>
    <div class="form-group">
        <label for="prenom">Prénom :</label>
        <input type="text" id="prenom" name="prenom" required pattern="[A-Za-zÀ-ÖØ-öø-ÿ\s]+">
        <div class="error" id="prenomError"></div>
    </div>
    <div class="form-group">
        <label for="email">Email :</label>
        <input type="email" id="email" name="email" required>
        <div class="error" id="emailError"></div>
    </div>
    <div class="form-group">
        <label for="dateAnniversaire">Date d'anniversaire :</label>
        <input type="date" id="dateAnniversaire" name="dateAnniversaire" required>
        <div class="error" id="dateAnniversaireError"></div>
    </div>
    <div class="form-group">
        <label for="adresse">Adresse :</label>
        <input type="text" id="adresse" name="adresse" required>
        <div class="error" id="adresseError"></div>
    </div>
    <button type="submit">Soumettre</button>
</form>
```

WINDOW

Lors de l'envoi du formulaire par la méthode GET, on envoie au travers de la requête HTTP de manière visible, les informations saisies dans notre formulaire.

Avec window.location, on obtient l'ensemble des propriétés disponibles.

- window représente notre page en cours
- location contient toutes les informations de la page courante



WINDOW.LOCATION.SEARCH

L'attribut search contient l'ensemble des paramètres envoyés au serveur sous la forme

- ?param=valeur&param=valeur…

Ensuite, on peut aisément récupérer cette donnée dans une variable et en extraire les informations dans un tableau.

```
> let param = window.location.search
< undefined
> console.info(param);
? nom=boebion&prenom=jerome&email=jero%40free.fr&dateAnniversaire=2025-03-10&adresse=centre+afpa+de+Pompey
```
VM469:1

RÉCUPÉRATION DES DONNÉES D'UNE URL

Maintenant que nous connaissons les éléments à cibler pour récupérer les données du formulaire, il nous reste à écrire un script JS.

Voici un exemple de récupération des données de manière basique au travers d'une URL.

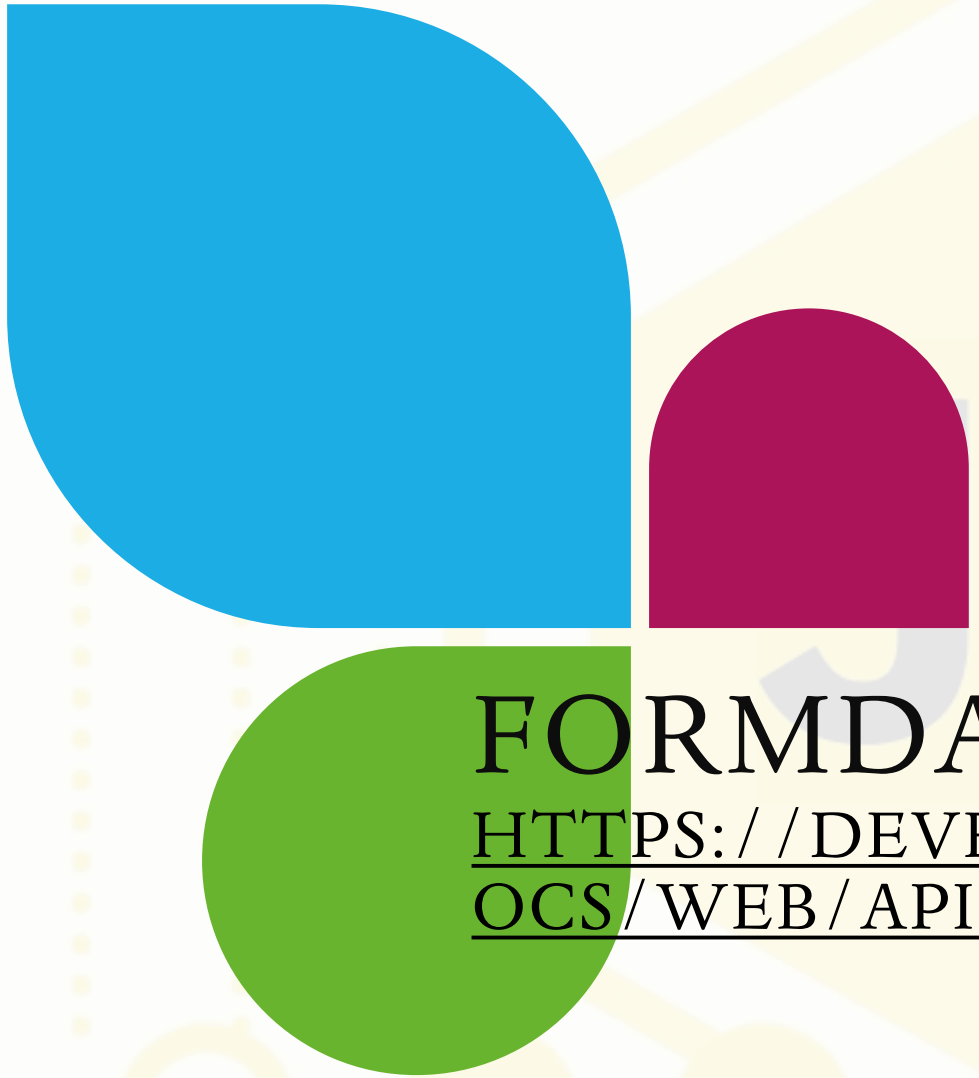
- Les tests dans ce code permettent de remplacer les codes ascii que le formulaire utilise lors de l'envoi.
  - *Dans cet exemple, pour l'email par exemple il remplace : par le code ascii correspondant %40 et les espaces par +*

```
function getParamsURL() {

    // objet contenant les parametres
    let varParams = {};
    // recupération du l'url en retirant le ?
    let search = window.location.search.substring(1);
    // contrôle
    console.info(search);

    // decompostion des parametre en retirant le &
    let varSearch = search.split('&');
    // contrôle
    console.info(varSearch);

    // parcours des élements et contruction de mon objet
    for (let i=0; i < varSearch.length; i++) {
        // suppression du =
        let parameter = varSearch[i].split('=');
        // mise en forme du caractère ASCII @
        if (parameter[0] === "email") {
            parameter[1] = parameter[1].replace('%40', '@');
        }
        // mise en forme du caractère ASCII +
        if (parameter[0] === "adresse") {
            parameter[1] = parameter[1].replaceAll('+', ' ');
        }
        // contruction de la réponse
        varParams[parameter[0]] = parameter[1].toUpperCase();

    }
    // controle
    console.info(varParams);

    return varParams;
}
```

AFFICHAGE

Ensuite dans la page de récupération, là où le formulaire nous redirige, nous n'avons plus qu'a afficher les données saisies.

```
document.addEventListener("DOMContentLoaded", function () {

    // appel de la fonction
    let params = getParamsURL();

    console.dir(params);

    // Mise en forme dans la page HTML
    document.getElementById("nom").textContent = params.nom;
    document.getElementById("prenom").textContent = params.prenom;
    document.getElementById("email").textContent = params.email;
    document.getElementById("dateAnniversaire").textContent = params.dateAnniversaire;
    document.getElementById("adresse").textContent = params.adresse;

});
```



FORMDATA

https://developer.mozilla.org/fr/docs/Web/API/FormData

FORMDATA

L'objet FormData a été standardisé et facilite grandement l'envoi vers un serveur.

- Il peut être utilisé indépendamment d'un formulaire, en lui ajoutant une à une les données à transmettre grâce à sa méthode append.
- Cette méthode prend en paramètres le nom et la valeur de la donnée ajoutée (clé/valeur).

Ensuite, on peut retrouver les données contenant dans notre objet FormData avec la méthode entries() qui retourne un itérator permettant d'accéder à l'ensemble des données.

Cet objet est souvent utilisé avec la méthode POST

```
document.getElementById('monFormulaire').addEventListener('submit', function(event) {
    event.preventDefault(); // Empêche l'envoi du formulaire

    // Crée un nouvel objet FormData à partir du formulaire
    const formData = new FormData(event.target);

    // Affiche les données du formulaire dans la console
    for (let [name, value] of formData.entries()) {
        console.log(`${name}: ${value}`);
    }
});
```

https://fr.javascript.info/formdata

MÉTHODES DISPONIBLES

Diverses méthodes sont disponibles avec formdata

https://developer.mozilla.org/fr/docs/Web/API/FormData

- append() : Ajoute une nouvelle valeur à une clé existante dans un objet FormData, ou ajoute la clé si elle n'existe pas encore.
- delete() : Supprime une paire clé/valeur d'un objet FormData.
- entries() : Renvoie un itérateur permettant de passer en revue toutes les paires clé/valeur contenues dans cet objet.
- get() : Renvoie la première valeur associée à une clé donnée à partir d'un objet FormData.
- getAll() : Renvoie un tableau de toutes les valeurs associées à une clé donnée à partir d'un objet FormData.
- has() : Renvoie un booléen indiquant si un objet FormData contient une certaine clé.
- key() : Renvoie un itérateur permettant de parcourir toutes les clés des paires clé/valeur contenues dans cet objet.
- set() : Renvoie un itérateur permettant de parcourir toutes les valeurs contenues dans cet objet.
- values() : Renvoie un itérateur permettant de parcourir toutes les valeurs contenues dans cet objet.

LOCALSTORAGE ET SESSION STORAGE

AU SEIN DU NAVIGATEUR

LOCALSTORAGE

- fonctionnalités JavaScript qui permet de stocker des données de manière persistante ou temporaire dans le navigateur web.
- Les données stockées dans localStorage restent disponibles même après la fermeture et la réouverture du navigateur.
- Les données stockées dans sessionStorage restent disponibles dans la session ou l'onglet du navigateur.
- utilisé pour stocker des données locales telles que des préférences utilisateur, des données de session ou des informations de configuration.
- accessibles uniquement via JavaScript et spécifiques au domaine et au protocole utilisés pour accéder à la page.

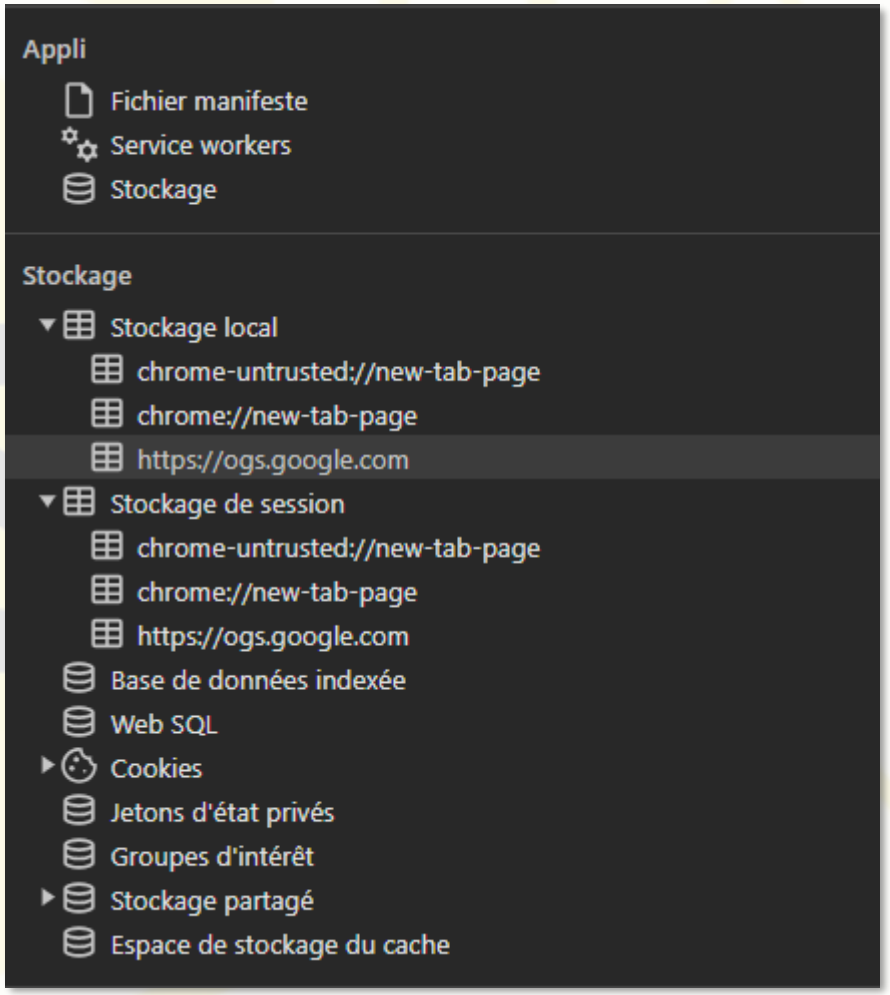
- Attention : En termes de sécurité web, l'utilisation doit être abordée avec prudence, car il peut présenter certains risques si les données sensibles sont mal gérées ou mal protégées.
  - Données sensibles : Ces données pourraient être accessibles à des scripts malveillants ou à d'autres attaques.
  - Injection de code : Ne stockez pas de données directement à partir d'entrées utilisateur sans validation et échappement appropriés. Cela pourrait permettre à des attaquants d'injecter du code malveillant dans les données stockées.
  - Taille limitée : il y a généralement une limite de taille (généralement quelques mégaoctets) par domaine. Stocker de grandes quantités de données peut affecter les performances du site web.
  - Utilisation sécurisée du HTTPS : Assurez-vous que votre site web utilise HTTPS pour chiffrer les données échangées entre le navigateur de l'utilisateur et votre serveur. Cela réduit le risque d'interception des données stockées.
  - Gestion des autorisations : Les navigateurs modernes demandent généralement la permission de l'utilisateur avant de stocker des données. Assurez-vous de respecter les préférences de l'utilisateur et de ne stocker que les données nécessaires.
  - Nettoyage des données : Évitez d'accumuler des données obsolètes ou inutiles Assurez-vous de nettoyer régulièrement les données qui ne sont plus nécessaires pour réduire les risques en cas de compromission.

DANS L'INSPECTEUR

Depuis l'inspecteur de votre navigateur, vous pouvez avoir accès aux différents espaces de stockage.

Depuis la console, vous pouvez également manipuler le localStorage ou la sessionStorage au travers des méthodes disponibles :

- getItem()
- removeItem()
- setItem()
- length()
- clear()



LOCALSTORAGE VS SESSIONSTORAGE

	localStorage	sessionStorage
Type d'objet	Objet JavaScript	
Portée	Domaine du site	Session de navigation
Durée de stockage	Persistant (reste jusqu'à ce qu'il soit explicitement supprimé)	Temporaire (disponible uniquement pendant la durée de la session)
Volume de stockage	Plusieurs Mo (selon le navigateur)	
Accessibilité	Tous les scripts de la même origine (domaine, protocole et port)	
Persistance	Persistant, reste jusqu'à ce qu'il soit explicitement supprimé	Temporaire, disparaît lorsque la session se termine (par exemple, lorsque l'onglet ou le navigateur est fermé)
Utilisation typique	Stockage de données utilisateur, telles que des préférences ou des données de configuration	Stockage de données temporaires nécessaires pendant la session de navigation de l'utilisateur, telles que des informations de session utilisateur ou des états temporaires
Sécurité	Doit être utilisé avec prudence pour éviter les fuites de données sensibles	
Exemple d'utilisation	Stockage des préférences utilisateur	Stockage des informations de session utilisateur

EXEMPLE

Reprenons notre exemple précédent en remplaçant la méthode basique par l'utilisation du formData et de localstorage

- Au niveau du formulaire, sur la soumission, je crée un élément formData à partir de mon formulaire.
- Ensuite, afin de pouvoir le transmettre, je dois effectuer une conversion en objet Javascript et le transformer en JSON (JSON.stringify).
  - *En effet, le localstorage ne peut contenir des objets complexes tel que formData.*

```
document
  .getElementById("userForm")
  .addEventListener("submit", function (event) {
    event.preventDefault();

    let isValid = checkForm();

    if (isValid) {
      // Stocker les informations dans le localStorage

      // creation du formData
      let formData = new FormData(event.target);
      // Affiche les données du formulaire dans la console
      for (let [name, value] of formData.entries()) {
        console.log(`${name}: ${value}`);
      }

      // Convertit les données du formulaire en objet
      // le localstorage ne peux stocker que du texte donc le JSON est parfait
      // Cela implique de transformer l'objet FormData en un objet JavaScript standard,
      // puis de le convertir en chaîne JSON avec JSON.stringify().
      const formDataObject = {};
      formData.forEach((value, key) => {
        formDataObject[key] = value;
      });


      // Sauvegarde les données dans localStorage
      localStorage.setItem("userInfo",JSON.stringify(formDataObject));

      // Rediriger vers la page de compte
      window.location.href = "compte.html";
    }
  });
```

EXEMPLE SUITE

Enfin sur la page de compte, je vais pouvoir récupérer les informations depuis le localstorage et reparcourir les données.

1. Lecture depuis le localstorage en fonction du nom donné et convertion en objet Javascript (JSON.parse)
2. Parcours de l'objet Javascript, pour l'afficher dans les bons éléments HTML.

```
// Récupérer les informations de l'utilisateur depuis le localStorage
// Lorsque vous récupérez les données depuis localStorage,
// vous devez les convertir de JSON en objet JavaScript avec JSON.parse().
const userInfoFormData = JSON.parse(localStorage.getItem("userInfo"));

// contrôle
console.info(userInfoFormData);

// si l'objet est bien présent dans le localstorage
// mise à jour des éléments html
if (userInfoFormData) {

  // parcours du formData
  for(const pair of Object.entries(userInfoFormData)) {

    // contrôle
    console.table(pair);

    // ciblage de l'element correspondant
    const tag = document.getElementById(pair[0]);
    // remplissage de la valeur de l'element
    tag.textContent = pair[1];
  }

} else {
  alert( 'Aucune donnée de formulaire trouvée.');
}
```

EXEMPLE DE CLASSE UTILITAIRE

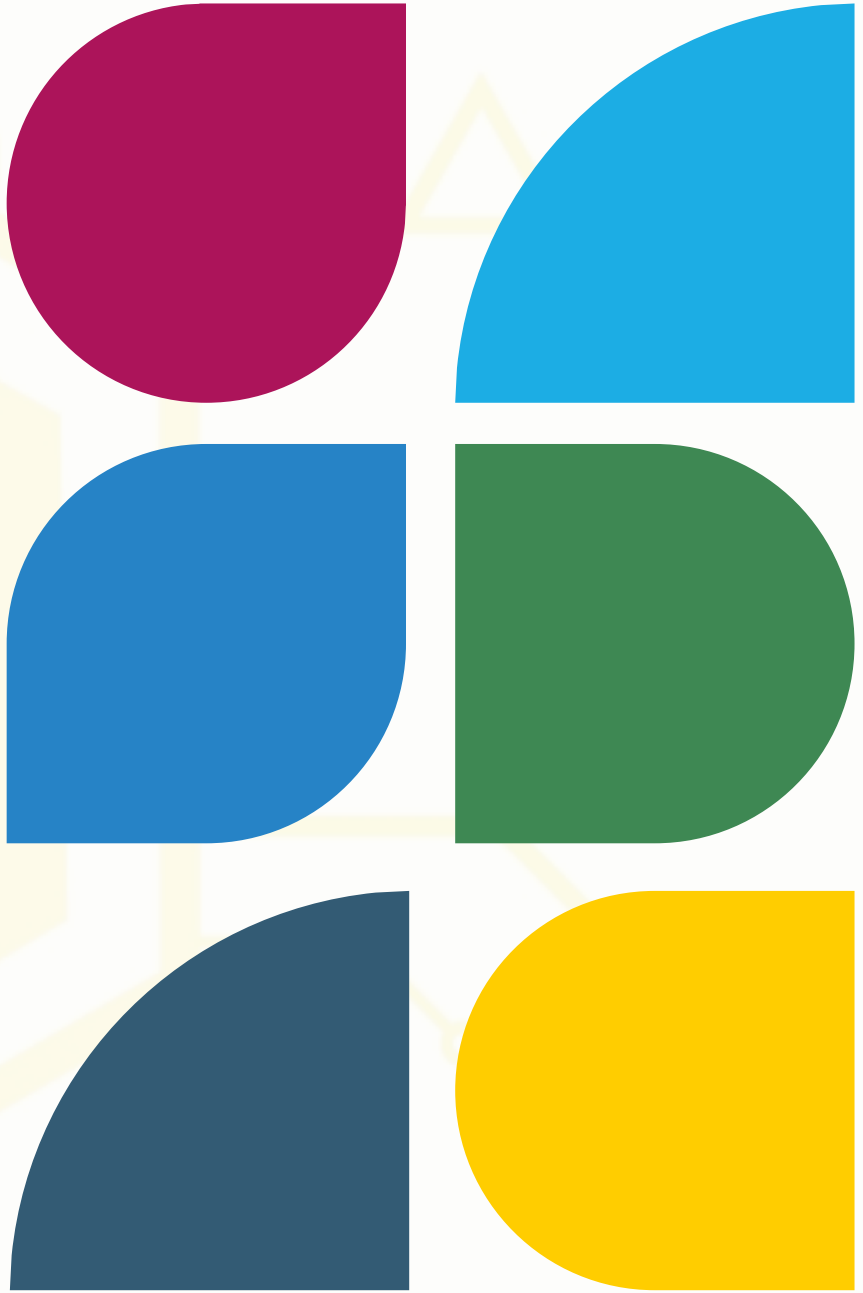
Voici un exemple de classe Javascript contenant des méthodes d'accès sur localStorage et sessionStorage.

Pour pouvoir utiliser ces méthodes, il suffit de les importer :

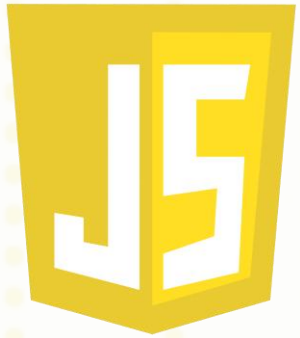
```
// Importer les fonctions du module de stockage
import {
    saveTolocalStorage,
    saveToSessionStorage,
    getAllLocalStorageItems,
    getAllSessionStorageItems,
    clearLocalStorage,
    clearSessionStorage
} from './storage-utils.js';
```

```
// Fonctions utilitaires pour la gestion du stockage
export function saveTolocalStorage(key, data) {
    localStorage.setItem(key, JSON.stringify({
        ...data,
        date: new Date().toLocaleString()
    }));
}

export function saveToSessionStorage(key, data) {
    sessionStorage.setItem(key, JSON.stringify({
        ...data,
        date: new Date().toLocaleString()
    }));
}

export function getAllLocalStorageItems() {
    return Object.keys(localStorage)
        .map(key => ({
            key,
            value: JSON.parse(localStorage.getItem(key))
        }));
}

export function getAllSessionStorageItems() {
    return Object.keys(sessionStorage)
        .map(key => ({
            key,
            value: JSON.parse(sessionStorage.getItem(key))
        }));
}

export function clearLocalStorage() {
    localStorage.clear();
}

export function clearSessionStorage() {
    sessionStorage.clear();
}
```

Afpa
se former, avancer



JavaScript



Coder en JavaScript.. Partie 2

Faire des requêtes vers des serveurs distants… API, gestion de l'asynchrone, gestion des erreurs

GESTION DES ERREURS

TRY..CATCH…FINALLY

Peu importe notre niveau en programmation, nos scripts comportent parfois des erreurs.

- Elles peuvent être dues à nos erreurs, à une entrée utilisateur imprévue, à une réponse erronée du serveur et à mille autres raisons.

Généralement, un script “meurt” (s’arrête immédiatement) en cas d’erreur, en l’affichant dans la console.

La structure try..catch est identique à celle vu avec Java.

```
try {

  // code...

} catch (err) {

  // Gestion des erreurs

}
```

En cas d’erreur, JavaScript génère un objet contenant les détails à son sujet. L’objet est ensuite passé en argument à catch

Pour toutes les erreurs intégrées, l’objet d’erreur a deux propriétés principales :

- Name : Nom de l’erreur. Par exemple, pour une variable non définie, il s’agit de "ReferenceError".
- Message : Message textuel sur les détails de l’erreur.

Il existe d’autres propriétés non standard disponibles dans la plupart des environnements. L’un des plus largement utilisés et supportés est :

- Stack : Pile d’exécution en cours : chaîne contenant des informations sur la séquence d’appels imbriqués ayant entraîné l’erreur. Utilisé à des fins de débogage.

CEPENDANT, IL FAUT TENIR COMPTE DES COMPORTEMENTS DE NOTRE CODE, DÛ À L'ASYNCHRONE. LE COMPORTEMENT DE TRY CATCH EST SYNCHRONE.

Si une exception se produit dans le code “planifié”, comme dans setTimeout, try...catch ne l’attrapera pas

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

C’est parce que la fonction elle-même est exécutée ultérieurement, lorsque le moteur a déjà quitté la structure try...catch.

Pour capturer une exception dans une fonction planifiée, try...catch doit être à l’intérieur de cette fonction.

```
setTimeout(function() {
  try {
    noSuchVariable; // try...catch gère l'erreur !
  } catch {
    alert( "error is caught here!" );
  }
}, 1000);
```

LEVER NOS PROPRES EXCEPTIONS

L'instruction throw génère une erreur.

La syntaxe est la suivante :

- throw <error object>

Techniquement, on peut utiliser n'importe quoi comme objet d'erreur.

- Cela peut même être une primitive, comme un nombre ou une chaîne, mais il est préférable d'utiliser des objets, de préférence avec les propriétés name et message (pour rester quelque peu compatibles avec les erreurs intégrées).

JavaScript comporte de nombreux constructeurs intégrés pour les erreurs standards : Error, SyntaxError, ReferenceError, TypeError et autres.

Nous pouvons également les utiliser pour créer des objets d'erreur.

```
let error = new Error(message);
// ou
error = new SyntaxError(message);
error = new ReferenceError(message);

error = new Error("Houston, we have a problem o_O");

alert(error.name); // Error
alert(error.message); // Houston, we have a problem o_O

// JSON.parse génére une SynaxError
try {
  JSON.parse("{ bad json o_O }");
} catch (err) {
  alert(err.name); // SyntaxError
  alert(err.message); // Unexpected token b in JSON at position 2
}

let json = '{ "age": 30 }'; // données incomplètes

try {

  let user = JSON.parse(json); // <-- pas d'erreurs

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name"); // (*)
  }

  alert( user.name );

} catch (err) {
  alert( "JSON Error: " + err.message ); // JSON Error: Incomplete data: no name
}
```

LES ERREURS INTÉGRÉES

En JavaScript, il existe plusieurs types d'erreurs intégrées qui peuvent être levées pendant l'exécution d'un programme.

Voici quelques-unes des erreurs les plus courantes :

1. EvalError : Indique une erreur concernant la fonction globale eval(). Cette erreur n'est plus levée par le moteur JavaScript moderne, mais elle existe toujours pour des raisons de compatibilité.
2. RangeError : Levée lorsqu'une valeur n'appartient pas à l'ensemble ou à la plage de valeurs autorisées. Par exemple, cela peut se produire lorsque vous essayez de créer un tableau avec une taille négative.
3. ReferenceError : Levée lorsqu'on fait référence à une variable qui n'existe pas.
4. SyntaxError : Levée lorsqu'il y a une erreur dans l'analyse du code JavaScript. Cela se produit souvent à cause de fautes de frappe ou de syntaxe incorrecte.
5. TypeError : Levée lorsqu'une valeur n'est pas du type attendu. Par exemple, l'appel d'une méthode sur une valeur null ou undefined.
6. URIError : Levée lorsqu'une fonction globale manipulant des URI est utilisée de manière incorrecte. Par exemple, encodeURI ou decodeURI.
7. AggregateError : Introduite dans ES2021, elle est utilisée pour regrouper plusieurs erreurs en une seule. Elle est souvent utilisée avec les promesses pour capturer plusieurs rejets.
8. InternalError : Levée lorsqu'une erreur interne se produit dans le moteur JavaScript. Ce type d'erreur est rarement vu par les développeurs, car il est généralement lié à des bugs dans le moteur JavaScript lui-même.

PROPAGER UNE EXCEPTION

Dans l'exemple précédent, nous utilisons try...catch pour gérer des données incorrectes.

Mais est-il possible qu'une autre erreur inattendue se produise dans le bloc try {...} ?

- Comme une erreur de programmation (variable is not defined) ou quelque chose d'autre, pas seulement cette "donnée incorrecte".

Catch ne doit traiter que les erreurs qu'il connaît et "renvoyer" toutes les autres.

La technique "rethrowing" peut être expliqué plus en détail comme :

- Catch obtient toutes les erreurs.
- Dans le bloc catch (err) {...} nous analysons l'objet d'erreur err.
- Si nous ne savons pas comment le gérer, nous faisons throw err.

```
let json = '{ "age": 30 }'; // données incomplètes
try {

  let user = JSON.parse(json);

  if (!user.name) {
    throw new SyntaxError("Incomplete data: no name");
  }

  blabla(); // erreur inattendue

  alert( user.name );

} catch (err) {

  if (err instanceof SyntaxError) {
    alert( "JSON Error: " + err.message );
  } else {
    throw err; // propager (*)
  }

}
```

AJAX

ASYNCHRONOUS JAVASCRIPT AND XML

AJAX

AJAX est une pratique de programmation qui consiste à construire des pages web plus complexes et plus dynamiques en utilisant une technologie connue sous le nom de XMLHttpRequest

https://developer.mozilla.org/fr/docs/Web/API/XMLHttpRequest .

AJAX vous permet de mettre à jour simplement des parties du DOM d'une page web HTML au lieu de devoir recharger la page entière.

AJAX vous permet également de travailler de manière asynchrone, c'est-à-dire que votre code continue à s'exécuter pendant que la partie de votre page web essaie de se recharger

- par opposition à la méthode synchrone qui bloque l'exécution de votre code jusqu'à ce que la partie de votre page web ait fini de se recharger.

Note : Avec les sites web interactifs et les standards modernes du web, AJAX est progressivement remplacé par des fonctions dans les cadres JavaScript et l'API standard officielle Fetch API.

FONCTIONNEMENT

https://developer.mozilla.org/fr/docs/Web/API/XMLHttpRequest

Il y a 5 étapes pour mettre en place une requête AJAX

1. On crée un nouvel objet XMLHttpRequest.
2. On appelle sa méthode open() afin de l'initialiser.
3. On ajoute un gestionnaire d'évènement pour son évènement load, qui se déclenchera lorsque la réponse sera reçue sans erreur.
   - Dans ce gestionnaire, on appelle la méthode initialize() avec les données.
4. On ajoute un gestionnaire d'évènement pour son évènement error, qui se déclenchera s'il y a une erreur avec la requête.
5. On envoie la requête.
6. On enveloppe tout ce code dans un bloc try…catch, afin de gérer les éventuelles erreurs déclenchées par open() ou send().

ajax.js > ...

```
1  const request = new XMLHttpRequest();
2
3  try {
4    request.open("GET", "products.json");
5
6    request.responseType = "json";
7
8    request.addEventListener("load", () => initialize(request.response));
9    request.addEventListener("error", () => console.error("Erreur XHR"));
10
11   request.send();
12 } catch (error) {
13   console.error(`Erreur XHR ${request.status}`);
14 }
15
```

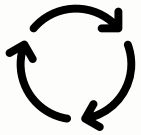
INTERROGER UN SERVEUR

L'objet XMLHttpRequest permet de créer des requêtes HTTP en JavaScript

- open permet de configurer la requête HTTP
  - Type de requête (GET, POST ou PUT)
  - url cible
  - Booléen indiquant si requête asynchrone ou non
- send envoie la requête HTTP
  - null en paramètre si GET
  - Les paramètres si POST
- responseText est la réponse du serveur sous forme textuel

Requêtes synchrones et requêtes asynchrones

- Pour gérer une requête asynchrone, on va utiliser les événements qui vont notifier de la disponibilité de la réponse.
- Un événement de type load indique la fin de traitement de la requête par le serveur



```
const requestURL = 'https://neojero.github.io/JsonApi/motif.json';

// Création de la requête
const request0 = new XMLHttpRequest();

try {
    // le 3ème paramètre indique en synchrone ou asynchrone
    request0.open("GET", requestURL, true);

    // par défaut, la requête réponds en texte
    request0.responseType = "json";

    // Ajout d'un écouteur pour l'événement "load"
    request0.addEventListener("load", () => {
        if (request0.status >= 200 && request0.status < 300) {
            console.info(request0.response);
        } else {
            console.error(`Erreur HTTP : ${request0.statusText}`);
        }
    });

    // Ajout d'un écouteur pour l'événement "error"
    request0.addEventListener("error", () => {
        console.error(`Erreur réseau lors de la requête`);
    });

    // Envoi de la requête
    request0.send();

} catch (error) {
    console.error(`Erreur inattendue : ${error.message}`);
}
```

LES PROPRIÉTÉS DE XMLHTTPREQUEST

En créant une connexion, nous avons alors accès à un ensemble de propriétés permettant de pouvoir traiter la réponse.

```
> console.warn(new XMLHttpRequest());
⚠ ▶ VM172:1
▼ XMLHttpRequest {onreadystatechange: null, readyState: 0, timeout: 0,
  withCredentials: false, upload: XMLHttpRequestUpload, …} i
    onabort: null
    onerror: null
    onload: null
    onloadend: null
    onloadstart: null
    onprogress: null
    onreadystatechange: null
    ontimeout: null
    readyState: 0
    response: ""
    responseText: ""
    responseType: ""
    responseURL: ""
    responseXML: null
    status: 0
    statusText: ""
    timeout: 0
  ▶ upload: XMLHttpRequestUpload {onloadstart: null, onprogress: null, o
    withCredentials: false
  ▶ [[Prototype]]: XMLHttpRequest
```

GESTION FORMAT JSON

Le langage JavaScript permet de gérer facilement ce format de données.

- La fonction JSON.parse permet de transformer une chaîne de caractères conforme au format JSON en un objet JavaScript.
- La fonction JSON.stringify joue le rôle inverse : elle transforme un objet JavaScript en chaîne de caractères conforme au format JSON.

```
var avion = {
  marque : "Airbus",
  couleur : "Rouge"
}

console.log("avion.marque " + avion.marque);
console.log("avion.couleur " + avion.couleur);

var textAvion = JSON.stringify(avion);
console.log("Stringify Avion " + textAvion);

var parseAvion = JSON.parse(textAvion);

parseAvion.couleur = "bleu";

console.log("Parse Avion.marque " + parseAvion.marque);
console.log("Parse Avion.couleur " + parseAvion.couleur);
```

```
avion Airbus
avion Rouge
Stringify {"marque":"Airbus","couleur":"Rouge"}
Parse Avion Airbus
Parse Avion bleu
```

LES PROMESSES

3…2….1…. !!

ASYNCHRONE

Par défaut, Javascript est un langage synchrone.

Dans un contexte Web, cela peut poser problèmes d'où l'utilisation de l'asynchrone afin de permettre de continuer l'exécution du code indépendamment de l'attente du résultat d'une autre méthode.

En JavaScript, nous allons utiliser beaucoup de fonctions de callback qui vont générer de l'attente pendant que le reste du programme va se poursuivre.

- Inconvénient : on ne peut pas prédire la fin de l'exécution de la fonction de callback et l'ordre des différentes fonctions surtout si on a plusieurs callbacks.

solution : les promesses !

Les promesses vont permettre d'attendre qu'une opération asynchrone ait réussit et soit terminée avant d'exécuter un autre bout de code.

LES PROMESSES

https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Using_Promises

Javascript intègre un nouvel outil depuis 2015, l'objet Promise qui permet d'utiliser l'asynchrone dans les scripts avec async et await

- Beaucoup d'API fonctionnent aujourd'hui sur ce principe et nous aurons que peu souvent à créer nos promesses.

Une promesse peut être :

- soit en cours (promis mais pas encore fait)
- soit honorée (promis et réalisé)
- soit rompue (on ne fait pas ce qu'on a promis et on a prévenu).

```
const promesse = new Promise((resolve, reject) => {
    // Appel de resolve() si la promesse est résolue
    resolve("Action réussie");
    // Appel de reject() si elle est rejetée */
    reject("Echec de l'opération");
});
```

Cet objet à travers son exécuteur reçoit deux arguments à savoir les fonctions resolve et reject.

Le paramètre "resolve" correspond à une fonction qui sera exécuté si nous considérons que l'opération dans la promesse s'est bien déroulée.

Le paramètre "reject" est également une fonction mais celle-ci est utilisée pour indiquer à l'utilisateur une erreur.

LES PROMESSES

Lorsque notre promesse est créée, celle-ci possède deux propriétés internes :

- Propriété state (état) dont la valeur :
  - «pending» (en attente)
  - «fulfilled» (promesse tenue ou résolue)
  - «rejected» (promesse rompue ou rejetée)
- Propriété result qui va contenir la valeur de notre choix.

Note : l'état d'une promesse une fois résolue ou rejetée est fini et ne peut pas être changé donc un seul résultat : **soit résolue, soit rejetée.**

Création d'une promesse

```
const direSalutation = function (salutation) {

    return new Promise( function(resolve, reject) {

        if(salutation === 'bonjour') {

          resolve('Vous avez dit ' + salutation);

        } else {

          reject('Vous n\'avez pas saluez');

        }
    })
}
```

UTILISER ET EXPLOITER UNE PROMESSE

Pour obtenir et exploiter le résultat d'une promesse, on va généralement utiliser la méthode then() et catch() du constructeur Promise.

```
promesse()
    .then(function (result) {
        // traitement de result
    })
    .catch(function (error) {
        // traitement de l'erreur
    });
```

Version fonction fléchée

```
promesse().then((result) => {
    // instruction pour le traitement du résultat
}).catch((error) => {
    // instruction pour le traitement d'erreurs
});
```

Utilisation d'une promesse

```
direSalutation("hello")
  .then(function(result) {

    console.log(result);

  })
  .catch(function(error) {

    console.log(error);

  });
```

EXEMPLE

La promesse rend un résultat qu'on peut passer en paramètre de notre resolve => resolve(result)

Ensuite, lors de la capture par then, on peut alors récupérer notre résultat pour un traitement.

Déclaration

```
let calculFctfleche = new Promise ( (resolve, reject) => {
    // ... code ... SQL, API etc...
    let result = 200*120;
    if (result) {
        resolve(result);
    } else {
        reject();
    }
});

let calcul = new Promise ( function (resolve, reject) {
    // ... code ... SQL, API etc...
    let result = 200*120;
    if (result) {
        resolve(result);
    } else {
        reject();
    }
});
```

Version fonction fléchée

Utilisation

```
// Utilisation de la promesse
calcul.then( function(result) {
    console.log('resultat : ' + result);
}).catch( function() {
    console.log('oops une erreur');
});


calculFctfleche.then( (result) => {
    console.log('resultat : ' + result);
}).catch( () => {
    console.log('oops une erreur');
});
```

EXEMPLE

```js
let MultplicationFctFlechee = function (num1, num2) {
    return new Promise ( function (resolve, reject) {
        // ... code ... SQL, API etc...
        let result = num1 * num2;
        if (result > 1000) {
            resolve(result);
        } else {
            reject('result trop petit');
        }
    });
}

let Multplication = (num1, num2) => {
    return new Promise ( (resolve, reject) => {
        // ... code ... SQL, API etc...
        let result = num1 * num2;
        if (result > 1000) {
            resolve(result);
        } else {
            reject('result trop petit');
        }
    });
}
```

On déclare ici une fonction fléchée avec des paramètres en entrée et qui va nous rendre une promesse avec un résultat

Ensuite, lors de l'appel de la fonction, on fournit les paramètres qui seront traités par la promesse… En cas de rejet, on capture l'erreur renvoyée par la promesse avec **catch(err)**

```js
// Utilisation de la promesse

calcul(10,20).then( function (result) {
    console.log('résultat' + result);
}).catch( (err) => {
    console.log('oops une erreur : ' + err);
});

calcul(10,20).then( (result) => {
    console.log('résultat' + result);
}).catch( (err) => {
    console.log('oops une erreur : ' + err);
});
```

CHAINAGE DE PROMESSES

Le chainage des promesses en Javascript permet d'attendre le résultat d'une promesse pour ensuite l'utiliser dans une autre promesse et ainsi de suite.

La première promesse est exécutée.
Au bout de 3 secondes la première méthode "then" est appelée.

- Cette méthode contient un "return" de la deuxième promesse qui est alors exécutée.

Au bout de 2 secondes la deuxième méthode "then" est exécutée.
Et ainsi de suite jusqu'à la quatrième promesse.

```
▶ Promise {<pending>}
Promesse1
Promesse2
Promesse3
Promesse4
>
```

```javascript
// Première promesse
var promesse1 = function () {
    return new Promise( function(resolve, reject) {
        setTimeout(function() {
            resolve('Promesse1');
        }, 3000);
    })
}

// Deuxième promesse
var promesse2 = function () {
    return new Promise( function(resolve, reject) {
        setTimeout(function() {
            resolve('Promesse2');
        }, 2000);
    })
}

// Troisième promesse
var promesse3 = function () {
    return new Promise( function(resolve, reject) {
        setTimeout(function() {
            resolve('Promesse3');
        }, 1000);
    })
}

// Quatrième promesse
var promesse4 = function () {
    return new Promise( function(resolve, reject) {
        setTimeout(function() {
            resolve('Promesse4');
        }, 500);
    })
}
```

```javascript
promesse1()
    .then(function(res) {
        console.log(res)
        return promesse2();
    })
    .then(function(res) {
        console.log(res)
        return promesse3();
    })
    .then(function(res) {
        console.log(res)
        return promesse4();
    })
    .then(function(res) {
        console.log(res);
    })
    .catch(function(resErreur) {
        console.log(resErreur);
});
```

REPRISE DE L'EXEMPLE HTTP

Si nous devions reprendre l'exemple de l'envoi HTTP, nous pourrions alors utiliser la promesse de cette manière.

```
// Utilisation de la fonction fetchData avec une Promise
fetchData(requestURL)
    .then(response => {
        console.info(response);
        let data = response.members;
        console.table(data);
    })
    .catch(error => {
        console.error(error.message);
    });
```

```
const requestURL = 'https://neojero.github.io/JsonApi/motif.json';

// Fonction qui retourne une Promise pour la requête
function fetchData(url) {
    return new Promise((resolve, reject) => {
        const request0 = new XMLHttpRequest();

        try {
            request0.open("GET", url, true);
            request0.responseType = "json";

            // Traitement de resolve
            request0.addEventListener("load", () => {
                if (request0.status >= 200 && request0.status < 300) {
                    resolve(request0.response);
                } else {
                    reject(new Error(`Erreur HTTP : ${request0.statusText}`));
                }
            });

            // traitement de reject
            request0.addEventListener("error", () => {
                reject(new Error(`Erreur réseau lors de la requête`));
            });

            request0.send(null);

        } catch (error) {
            reject(new Error(`Erreur inattendue : ${error.message}`));
        }
    });
}
```

GÉRER LE CHAINAGE DE PROMESSES

Vous l'aurez compris, l'impact sur le code est de bloquer l'exécution et éviter la gestion d'opération en parallèle, lorsque ces dernières n'ont pas besoin de l'être.

Mais il faut faire attention, si vous rendez synchrone des opérations qui auraient pu se dérouler en parallèle, alors vous pourrez ralentir votre exécution de manière significative.

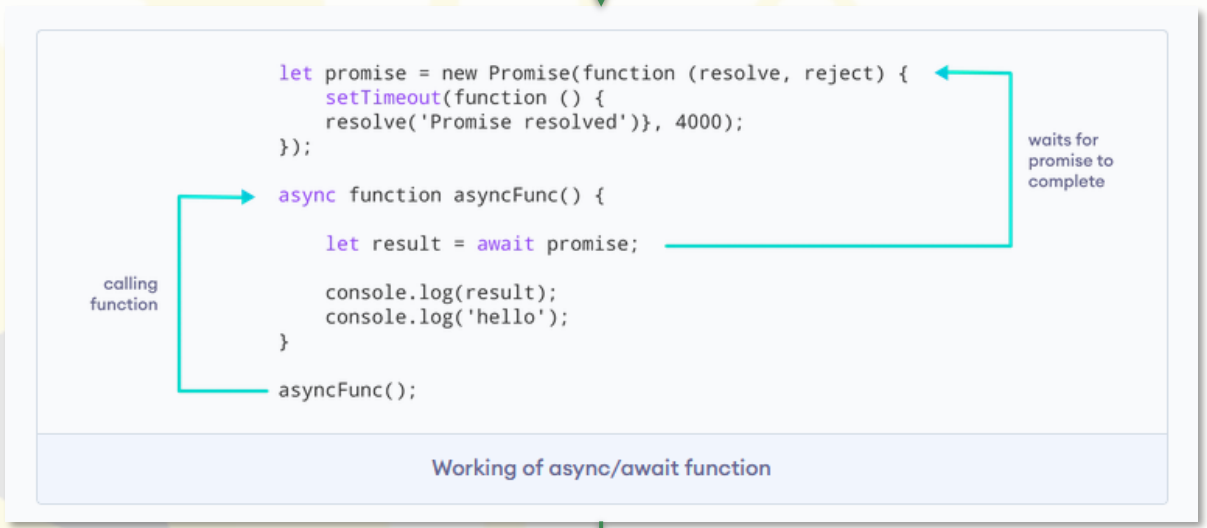
Solution : Promise.all()

```
//  ...
const promises = [];
for(let i=0 ; i<25 ; i++){
  promises.push(t()); //non-blocking execution
}
await Promise.all(promises);
// ...
```

```
// sans await
try {
    signup.then(function (data) {
        let user = data;
    }).catch((e)=>{
        let error = e;
    });

    while(!user && !error){
      // waiting loop pour attendre la fin de la promesse
    }
    return user;
} catch(e){
    // traitement du rejet
}
// version fléchée
try {
    signup.then((data)=>{
        let user = data;
    }).catch((e)=>{
        let error = e;
    });

    while(!user && !error){
      // waiting loop pour attendre la fin de la promesse
    }
    return user;
} catch(e){
    // traitement du rejet
}

// avec await
try {
    const user = await signup();
    return user;
} catch(e) {
    // traitement
}
```

EXEMPLE

```
// a promise
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {

    // wait until the promise resolves
    let result = await promise;

    console.log(result);
    console.log('hello');
}

// calling the async function
asyncFunc();
```



Output

```
Promise resolved
hello
```

LA BOUCLE FOR AWAIT

Il est possible de parcourir des tableaux, des chaînes de caractères, des collections de nœuds DOM… de manière asynchrone.

Comme pour le mot-clé await, la structure for await ... of ne peut être utilisée que dans une fonction asynchrone.

```
// boucle for await
async function afficherNombre(...values) {
  for await (let v of values) {
    console.log(v);
  }
}

// Si nous appelons deux fois de suite cette fonction :

afficherNombre(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
afficherNombre(101, 102, 103, 104, 105, 106, 107, 108, 109, 110);

/*
Alors nous verrons bien le comportement asynchrone puisque
l'affichage entremêlera les traitements des deux tableaux de valeurs
*/
```

MÉTHODES DISPONIBLES

Il existe d'autres méthodes disponibles avec l'objet Promise.

JavaScript Promise Methods

There are various methods available to the Promise object.

Method	Description
`all(iterable)`	Waits for all promises to be resolved or any one to be rejected
`allSettled(iterable)`	Waits until all promises are either resolved or rejected
`any(iterable)`	Returns the promise value as soon as any one of the promises is fulfilled
`race(iterable)`	Wait until any of the promises is resolved or rejected
`reject(reason)`	Returns a new Promise object that is rejected for the given reason
`resolve(value)`	Returns a new Promise object that is resolved with the given value
`catch()`	Appends the rejection handler callback
`then()`	Appends the resolved handler callback
`finally()`	Appends a handler to the promise

API FETCH

LE REMPLACEMENT DE XMLHTTPREQUEST

API FETCH

Elle permet d'utiliser JavaScript depuis une page pour construire et envoyer une requête HTTP à un serveur afin de récupérer des données.

Lorsque le serveur répond en fournissant les données, le code JavaScript peut les utiliser afin de mettre à jour la page, généralement en utilisant les API de manipulation du DOM.

Les données sont généralement demandées au format JSON, mais il peut tout aussi bien s'agir de HTML ou de texte.

Cette méthode est employée largement par les sites utilisant de nombreuses données tels que Amazon, YouTube, eBay, etc.

Avec ce modèle :

- Les mises à jour des pages sont plus rapides et il n'est plus nécessaire d'attendre un rechargement de la page : le site apparaît alors comme plus rapide et réactif.
- Il y a moins de données téléchargées pour chaque mise à jour, ce qui signifie une consommation moindre de la bande passante.
- Si cela n'était pas vraiment un problème sur un ordinateur de bureau avec une connexion à très haut débit, cela pouvait vite freiner la navigation sur les appareils mobiles et/ou aux endroits où l'accès à Internet est moins rapide.

FETCH : ENSEMBLE D'OBJET ET DE FONCTIONS AFIN D'EXÉCUTER DES REQUÊTES HTTP DE MANIÈRE ASYNCHRONE. IL PERMET D'EXÉCUTER DES REQUÊTES HTTP SANS RECHARGEMENT DU NAVIGATEUR.

Envoi d'une requête de type GET

```
fetch("url")
  .then(function(res) {
    if (res.ok) {
      return res.json();
    }
  })
  .then(function(value) {
    console.log(value);
  })
  .catch(function(err) {
    // Une erreur est survenue
  });
```

Envoi d'une requête de type POST

Content-Type et Accept : indique au service Web, le type de données qu'on envoie
body : les données à envoyer.

```
fetch("url", {
    method: 'POST',
    headers: {
        'Accept': 'application/json',
        'Content-Type': 'application/json'
    },
    body: JSON.stringify(jsonBody)
});
```

EXEMPLE

Récapitulons ce que fait ce fragment de script.

on utilise la fonction globale fetch() qui est le point d'entrée de l'API Fetch. Cette fonction prend l'URL comme paramètre

Ensuite, fetch() est une API asynchrone qui renvoie une promesse.

Comme fetch() renvoie une promesse, nous passons une fonction à la méthode then(). Cette méthode sera appelée lorsque le navigateur aura reçu une réponse du serveur pour la requête HTTP.

response.json() est également asynchrone

On passe une fonction à la méthode then() de cette nouvelle promesse. Cette fonction sera appelée lorsque le json sera prêt.

Enfin, on chaîne un gestionnaire catch() pour intercepter toute erreur qui serait déclenchée dans l'une des fonctions asynchrones ou des gestionnaires associés.

```
const requestURL = 'https://neojero.github.io/JsonApi/motif.json';

// Utilisation de l'API fetch pour récupérer les données
fetch(requestURL)
    // FETCH renvoie une promesse.
    .then(response => {
        // le gestionnaire leve une erreur si la requête a échoué
        if (!response.ok) {
            throw new Error(`Erreur HTTP : ${response.statusText}`);
        }
        // Sinon, en cas de réussite, la réponse est traité au format json
        // et retourne la réponse
        return response.json(); // Analyse la réponse en JSON
    })
    // traitement de la réponse
    .then(data => {
        console.info(data);
        let members = data.members;
        console.table(members);
    })
    .catch(error => {
        console.error(error.message);
    });
```

ASYNC ET AWAIT

PROGRAMMER DE L'ASYNCHRONE AVEC LE SYNCHRONE

CONCEPT

async et await sont des fonctionnalités introduites dans ES2017 pour simplifier le travail avec les Promesses et permettre d'écrire du code asynchrone de manière plus lisible et synchrone.

- Une fonction async retourne toujours une Promesse
- await ne peut être utilisé qu'à l'intérieur d'une fonction async
- await met en pause l'exécution de la fonction jusqu'à ce que la Promesse soit résolue
- La gestion des erreurs se fait préférentiellement avec try/catch

- fonction async de base

```
async function fetchUserData() {
  // Une fonction async peut utiliser await
  // Elle retourne automatiquement une Promesse
  return "Données utilisateur";
}

// Utilisation
fetchUserData().then(data => {
  console.log(data); // "Données utilisateur"
});
```

```
async function recupererDonnees() {
  try {
    // Simulation d'un appel API
    const reponse = await fetch('https://api.example.com/donnees');
    const donnees = await reponse.json();
    return donnees;
  } catch (erreur) {
    console.error('Erreur lors de la récupération:', erreur);
  }
}
```

LA PRINCIPALE DIFFÉRENCE AVEC LES PROMESSES CLASSIQUES

Sans async/await

```
function ancien() {
  return fetch('url')
    .then(reponse => reponse.json())
    .then(donnees => console.log(donnees))
    .catch(erreur => console.error(erreur));
}
```

Avec async/await

```
async function nouveau() {
  try {
    const reponse = await fetch('url');
    const donnees = await reponse.json();
    console.log(donnees);
  } catch (erreur) {
    console.error(erreur);
  }
}
```

EXEMPLE

Transformation de l'exemple avec l'utilisation de async et await

```
const requestURL = 'https://neojero.github.io/JsonApi/motif.json';

// Fonction asynchrone pour récupérer les données
async function fetchData() {
    try {
        const response = await fetch(requestURL);

        if (!response.ok) {
            throw new Error(`Erreur HTTP : ${response.statusText}`);
        }

        const data = await response.json(); // Analyse la réponse en JSON
        console.info(data);

        let members = data.members;
        console.table(members);

    } catch (error) {
        console.error(error.message);
    }
}

// Appel de la fonction asynchrone
fetchData();
```

EXEMPLE SELON LES DIFFÉRENTS APPELS

EXEMPLE 1

Appel d'une API synchrone

```javascript
// Exemple 1 : Appel API Synchrone (Bloquant)
function fetchUserDataSync() {
    // Simulation d'un appel API synchrone bloquant
    function syncXHR() {
        const xhr = new XMLHttpRequest();
        xhr.open('GET', 'https://jsonplaceholder.typicode.com/users/1', false); // false pour synchrone
        xhr.send(null);

        if (xhr.status === 200) {
            return JSON.parse(xhr.responseText);
        } else {
            throw new Error('Erreur de requête synchrone');
        }
    }

    console.log("Début de la requête synchrone");
    try {
        const userData = syncXHR();
        console.log("Données utilisateur (synchrone):", userData);
    } catch (error) {
        console.error("Erreur synchrone:", error);
    }
    console.log("Fin de la requête synchrone");
}
```

EXEMPLE 2

Appel d'une API avec promesses

```
// Exemple 2 : Appel API avec Promesses
function fetchUserDataPromise() {
    console.log("Début de la requête avec Promesses");

    fetch('https://jsonplaceholder.typicode.com/users/1')
        .then(response => {
            if (!response.ok) {
                throw new Error('Erreur de requête');
            }
            return response.json();
        })
        .then(userData => {
            console.log("Données utilisateur (Promesses):", userData);
        })
        .catch(error => {
            console.error("Erreur avec Promesses:", error);
        })
        .finally(() => {
            console.log("Requête Promesses terminée");
        });

    console.log("Après le début de la requête (Promesses)");
}
```

EXEMPLE 3

Appel d'une API avec async et Await

```
// Démonstration des différentes méthodes
function demonstrateurAppelsAPI() {
    console.log("=== Appel Synchrone ===");
    fetchUserDataSync();

    console.log("\n=== Appel avec Promesses ===");
    fetchUserDataPromise();

    console.log("\n=== Appel avec Async/Await ===");
    fetchUserDataAsyncAwait();
}

// Exécution de la démonstration
demonstrateurAppelsAPI();
```

```
// Exemple 3 : Appel API avec Async/Await
async function fetchUserDataAsyncAwait() {
    console.log("Début de la requête Async/Await");

    try {
        const response = await fetch('https://jsonplaceholder.typicode.com/users/1');

        if (!response.ok) {
            throw new Error('Erreur de requête');
        }

        const userData = await response.json();
        console.log("Données utilisateur (Async/Await):", userData);
    } catch (error) {
        console.error("Erreur Async/Await:", error);
    } finally {
        console.log("Requête Async/Await terminée");
    }

    console.log("Après la requête Async/Await");
}
```

BILAN

Appel Synchrone (Bloquant)

Problèmes :

- Bloque complètement l'exécution du script
- Interface utilisateur devient non réactive
- Mauvaise expérience utilisateur
- Non recommandé pour les applications modernes

Appel avec Promesses

Avantages :

- Non bloquant
- Gestion des succès et erreurs avec .then() et .catch()
- Chaînage possible des opérations
- Code plus lisible que les callbacks traditionnels

BILAN

Appel avec Async/Await

Avantages :

- Syntaxe proche du code synchrone
- Plus lisible et plus proche du style de programmation séquentiel
- Gestion des erreurs avec try/catch standard
- Facilite la lecture et l'écriture de code asynchrone complexe

Quelques points importants à noter

- fetch() retourne une Promesse
- Async/Await est du sucre syntaxique par-dessus les Promesses
- Les erreurs sont gérées différemment selon l'approche
- Le code asynchrone permet une meilleure réactivité et performance

ALLER PLUS LOIN
LIBRAIRIE ET FRAMEWORK

LIBRAIRIE - FRAMEWORK

De nombreux outils ont été créés pour faciliter la vie du développeur JavaScript et éviter de réinventer la roue à chaque nouveau projet :

- Les librairies (ou bibliothèques), qui complètent le langage standard pour faire gagner du temps ou améliorer la qualité du code.
- Les Framework, qui fournissent un ensemble de services de base formant le squelette de l'application.

- La librairie JavaScript la plus connue est sans aucun doute jQuery.
  - Créée à une époque où les divergences entre navigateurs étaient bien plus nombreuses qu'aujourd'hui, elle reste utilisée dans de très nombreux projets web.
  - Aujourd'hui jQuery tends à disparaître. Bootstrap a par exemple retiré jQuery de son framework.
- L'utilisation d'un Framework devient véritablement intéressante à partir d'un certain niveau de complexité de l'application :
  - Parmi la pléthore de Framework JavaScripts existants.
  - Angular, Ember, VueJs ou encore React (qui est plutôt une librairie qu'un framework, mais jouit d'une popularité croissante à l'heure actuelle).

JQUERY

Avec quelle technologie le remplacer ?

1. API FETCH : L'un des usages de JQuery est de faire des requêtes HTTP. L'API fetch() fait maintenant complètement parti des navigateurs, elle permet de faire des requêtes HTTP et vu que c'est intégré au moteur des navigateurs pas besoin d'importer d'autres librairies, c'est top !
2. Le JavaScript natif pour changer le contenu de la page. Désormais disponible directement depuis JavaScript (innerHTML, Evénements etc..)
3. STIMULUS offre des possibilités. permet notamment via le HTML de suivre beaucoup d'événements sans se prendre la tête et sans écrire des millions de lignes de code (inutile). Pour aller plus loin, jetez un œil à Stimulus Use, avec plein d'événements, faciles à importer, sans prise de tête.
   - https://stimulus.hotwired.dev/
4. AXIOS est un client HTTP, basée sur les promesses, compatible avec node.js et les navigateurs.
   - https://axios-http.com/fr/docs/intro

MERCI !

Jérôme BOEBION
Concepteur Développeur d'Applications
Version 4 - révision 2024

Afpa
se former, avancer