Afpa

se former, avancer

JAVASCRIPT

Coder en JavaScript..

AFPA CENTRE DE POMPEY

Introduction

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

Version actuelle : **ECMAScript 2025**

https://www.w3schools.com/Js/js_2025.asp

Rappel

Attention : Java et Javascript sont radicalement différent.

Script

Le JavaScript est un langage de script basé sur la norme ECMAScript.

Extension .js

Intégré

Il s'insère dans le code HTML d'une page web, et permet d'en augmenter le spectre des possibilités (interactivité et dynamisme).

POO

Ce langage de POO, faiblement typé, est exécuté côté client.
Mais également côté serveur avec Node.js

Normalisé

Normalisé par ECMAScript
https://262.ecma-international.org/

Où se place le code JavaScript ?

Integration de 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>
```

Le débogage

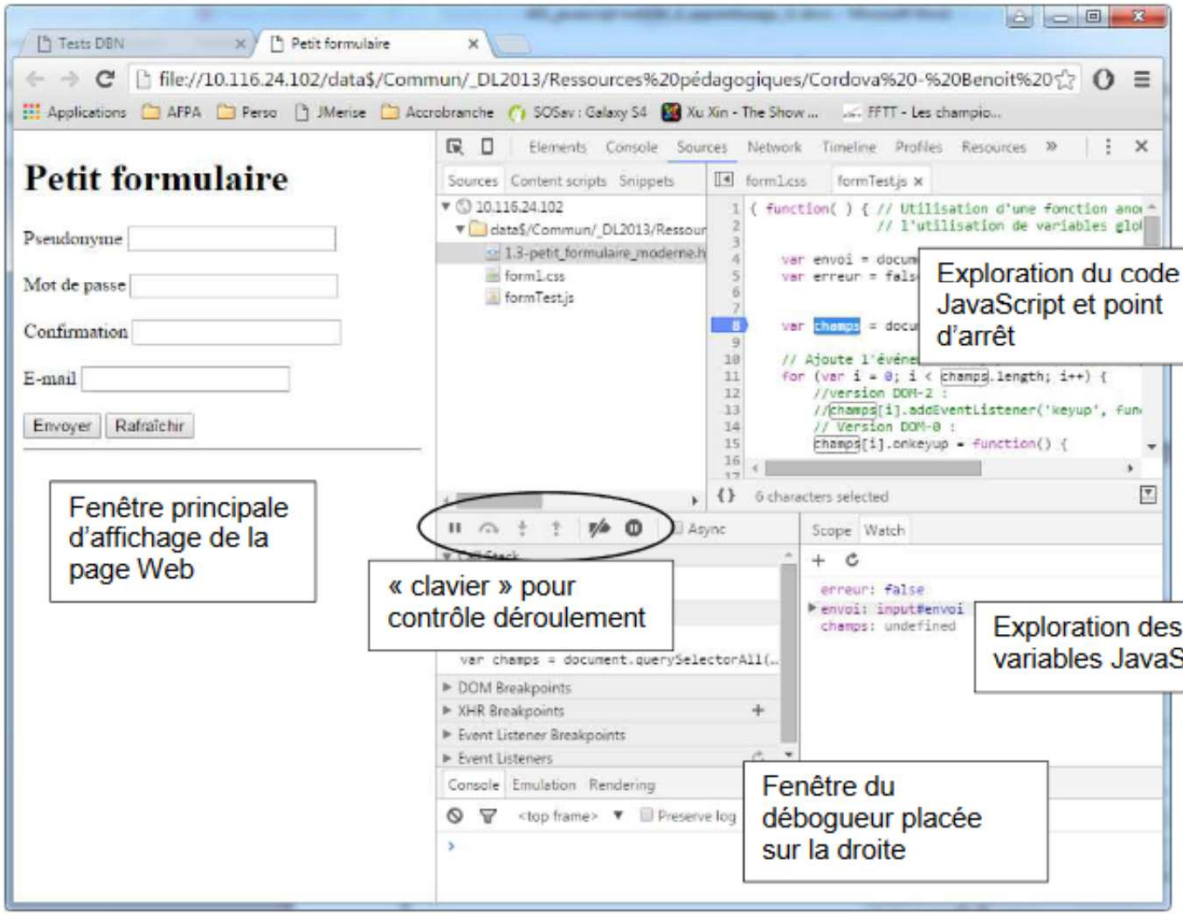
Debugger un script Javascript

En cas d'anomalie :

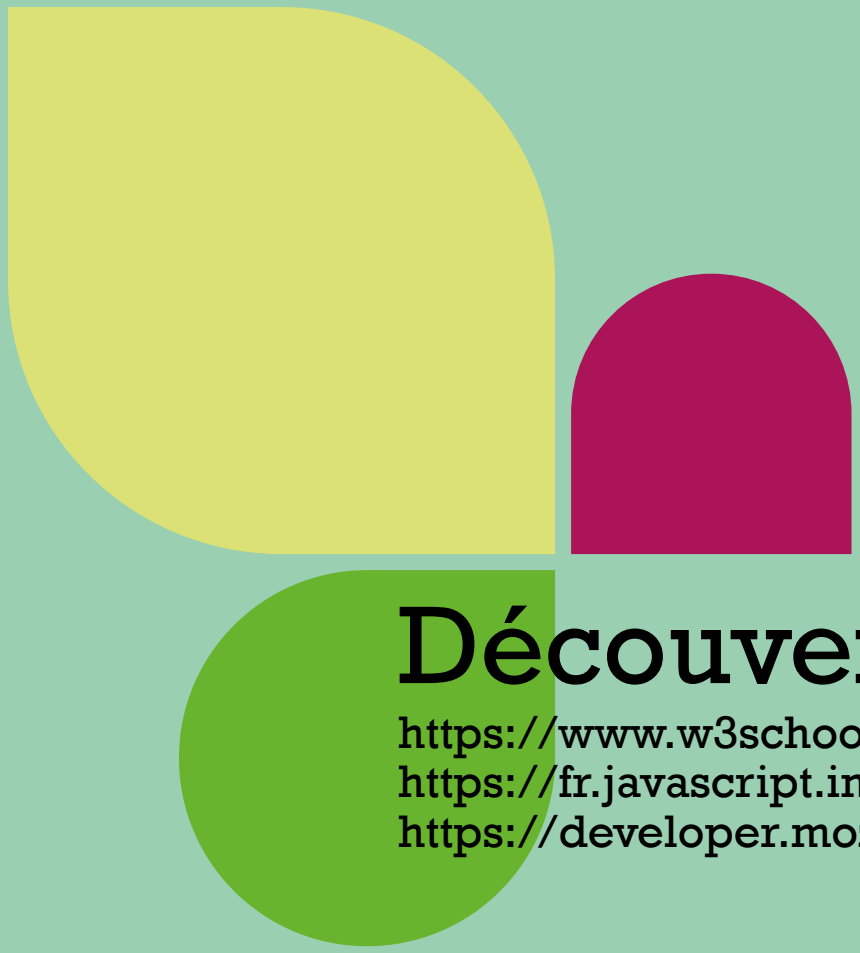
- Une erreur de syntaxe Javascript, c'est tout le chargement du bloc qui est annulé !
- L'utilisation alert() pour afficher des boites de dialogue supplémentaires permettant de tracer le déroulement d'un script.
- console.log() pour contrôler un ensemble de valeurs et les afficher dans la console du 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, …



Pour avancer pas à pas après une pause sur un point d'arrêt, utiliser les touches de fonction F10 et F11.



Découverte du langage

https://www.w3schools.com/js/
https://fr.javascript.info/
https://developer.mozilla.org/fr/docs/Web/JavaScript

Découverte du langage

Faisons connaissance…

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

Base

- Sensible à la case : alert() et non Alert()

// commentaire
bloc de commentaire /* … */

Convention

; pour terminer une instruction.
Même si c'est possible sans, cela permet d'éviter les erreurs.

Les variables

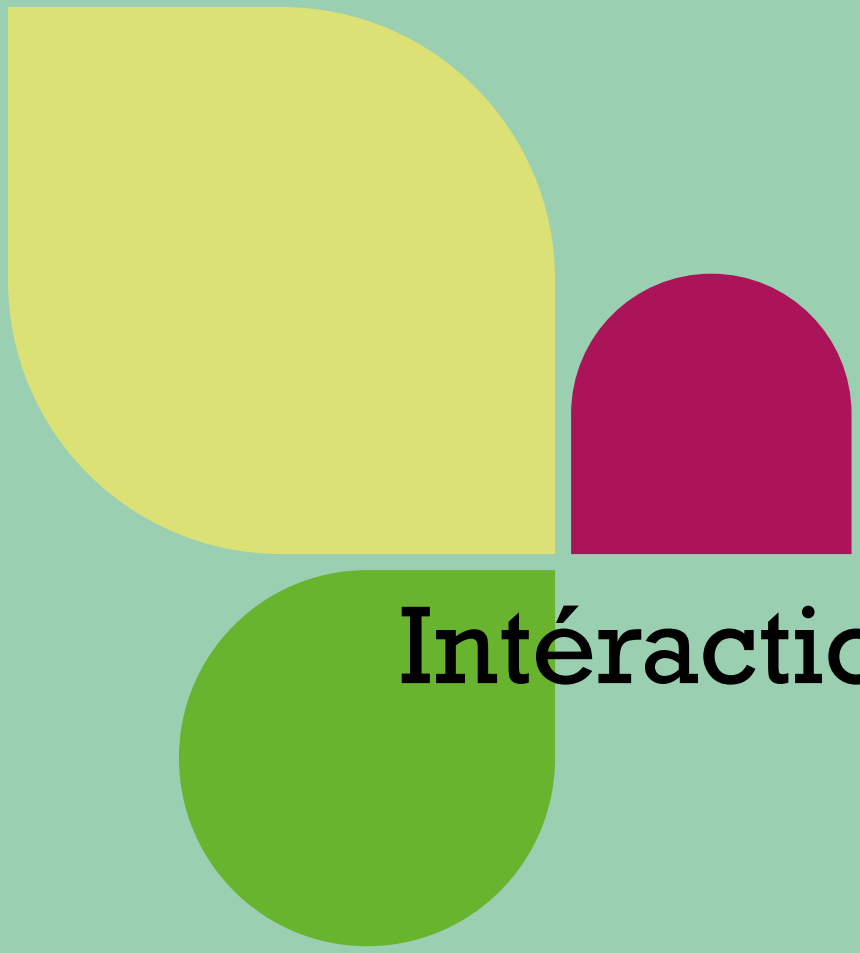
- Typée dynamiquement et à typage faible.
- Mutabilité des variables.
- Une variable non déclarée est **undefined**

var ou let

Attention : vous pourrez croiser le mot clé var plutôt que let. Pour l'instant considérer var comme l'ancienne version de let
let nom = "Jérôme"

tips

Pour les anciens navigateurs, on doit encapsuler notre code entre les balises de commentaires HTML (voir capture)



Intéractions

Intéraction

voyons quelques fonctions pour interagir avec l'utilisateur : alert, prompt et confirm.

alert : affiche un message model et attend que l'utilisateur appuie sur ok

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

confirm : affiche une fenêtre modale avec une question et deux boutons : OK et Annuler. Le résultat est true si vous appuyez sur OK et false dans le cas contraire.

let, var et const

déclaration de variables

Let, 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 ES6 : ECMAScript 2015 *(la plus grande mise à jour de JS)* , deux nouveaux mots-clés sont apparus : let et const.

1

Stocké en global ?

- var : Oui ✔ Une variable déclarée avec var en dehors d'une fonction est ajoutée à l'objet global (window dans les navigateurs).
- let : Non ❌ Une variable déclarée avec let en dehors d'un bloc ou d'une fonction n'est pas ajoutée à l'objet global, mais elle est accessible globalement dans le script.
- const : Non ❌ Comme let, const n'est pas ajouté à l'objet global, mais est accessible globalement dans le script.

Let, var et const

Précision :

La phrase "La portée (ou scope) d'une fonction est la seule à mettre toutes les variables sur un même pied d'égalité" est un peu trompeuse.

En réalité, var se comporte différemment de let et const même dans une fonction, car var n'est pas limité aux blocs (comme les boucles for, if, etc.).

2

Se limite à la portée d'une fonction ?

- var : Oui ✔ var est limité à la portée de la fonction où il est déclaré. Si déclaré en dehors de toute fonction, il est global.
- let : Oui ✔ let est limité à la portée du bloc (ou de la fonction) où il est déclaré.
- const : Oui ✔ const est aussi limité à la portée du bloc (ou de la fonction) où il est déclaré.

Let, var et const

Attention, une référence constante ne veut pas dire que la valeur derrière la référence est "immutable", Cela ne veut PAS dire que la valeur stockée est immuable.

3

Un bloc d'instruction

- var : Non ❌ Une variable déclarée avec var dans un bloc (comme un for, if, etc.) n'est pas limitée à ce bloc, elle est accessible dans toute la fonction ou globalement si elle est déclarée en dehors d'une fonction.
- let : Oui ✔ let est limité au bloc où il est déclaré.
- const : Oui ✔ const est aussi limité au bloc où il est déclaré.

Let, var et const

Réassigné et redéclaré ?

4

Peut être réassigné ?

- var : Oui ✔️
- let : Oui ✔️
- const : Non ❌ Une variable déclarée avec const ne peut pas être réassignée après son initialisation.

Peut être redéclaré ?

- var : Oui ✔️ var peut être redéclaré dans le même scope.
- let : Non ❌ let ne peut pas être redéclaré dans le même scope.
- const : Non ❌ const ne peut pas être redéclaré dans le même scope.

Hoisting

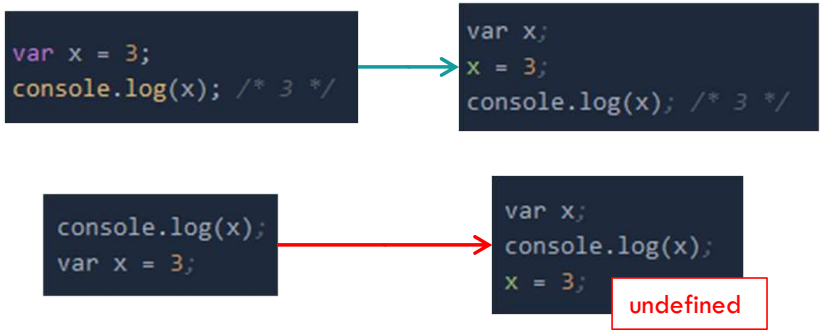
Cette mécanique consiste donc à faire virtuellement remonter la déclaration d'une variable (ou d'une fonction) tout en haut de son scope lors de l'analyse du code par le moteur d'interprétation Javascript.

- 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.

5

Est affecté par le hoisting ?

- var : Oui ✔ Les variables déclarées avec var sont "hoistées" (remontées) en haut de leur scope et initialisées avec undefined.
- let : Non ❌ Les variables déclarées avec let sont "hoistées" mais ne sont pas initialisées, ce qui crée une "Temporal Dead Zone" (TDZ) jusqu'à leur déclaration.
- const : Non ❌ Comme let, const est "hoisté" mais n'est pas initialisé, ce qui crée aussi une TDZ.



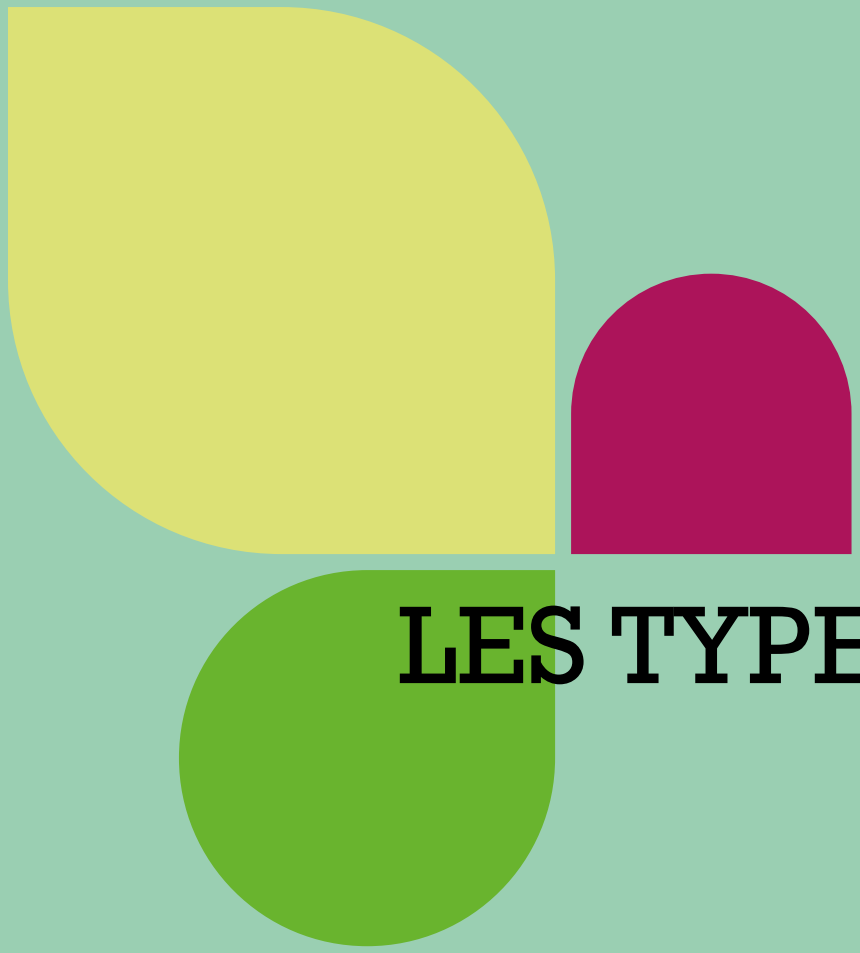
Conclusion

Règles simples pour éviter les problèmes

1. Déclare toujours tes variables avant de les utiliser.

2. Utilise const quand la valeur ne change pas.

3. Utilise let pour les valeurs qui évoluent. Évite var sauf cas très particulier *(code legacy = ancien code).*



LES TYPES

Les types

Primitif

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

Chaînes de caractères.	String
Nombres (entiers, décimaux, NaN, Infinity) et les grands entiers	Number bigint
Valeur vide volontaire.	null
Vrai ou faux.	boolean
Valeur d'une variable non initialisée.	undefined
Valeur unique et immutable, souvent utilisées comme clés d'objet. https://fr.javascript.info/symbol	symbol

Les types

Non primitif

Objet

Structure clé/valeur ou instances plus complexes.

Les objets incluent :

Array
Function
Date
RegExp
Map / Set
WeakMap / WeakSet
Error

typeof() permet de vérifier le type en cours.

… et tous les objets personnalisés.

⚠️ En JavaScript, *tout ce qui n'est pas un type primitif est un objet.*

Les opérateurs :

On retrouve la plupart des opérateurs que nous connaissons dans le développement.

Mais il existe quelques particularités !!

L'opérateur de coalescence des nuls '??'

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

Le résultat de a ?? b est :

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

Nous pouvons également utiliser une séquence de ?? pour sélectionner la première valeur dans une liste qui n'est pas null/undefined.

Disons que nous avons les données d'un utilisateur dans les variables firstName, lastName ou nickName. Tous peuvent être indéfinis, si l'utilisateur décide de ne pas entrer de valeurs correspondantes.

Nous aimerions afficher le nom d'utilisateur à l'aide de l'une de ces variables, ou afficher "Anonyme" si toutes sont null/undefined.

Utilisons l'opérateur ?? pour cela :

```
let firstName = null;
let lastName = null;
let nickName = "Supercoder";

// affiche la première valeur définie :
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder

// ancienne maniere de l'ecrire
alert(firstName || lastName || nickName || "Anonymous"); // Supercoder
```

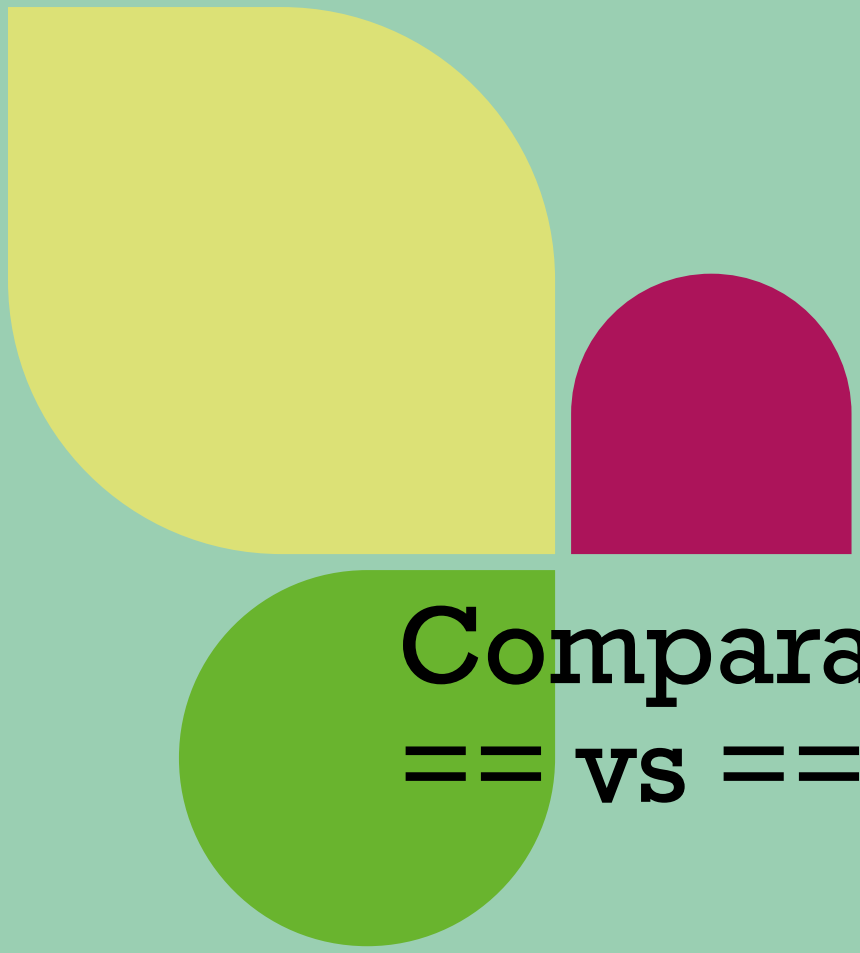
Historiquement, l'opérateur OR || était là en premier. Il existe depuis le début de JavaScript, donc les développeurs l'utilisaient à de telles fins depuis longtemps.

D'un autre côté, l'opérateur de coalescence des nuls ?? n'a été ajouté à JavaScript que récemment, et la raison en était que les gens n'étaient pas tout à fait satisfaits de ||.

La différence importante entre eux est que :

- || renvoie la première valeur *vraie*.
- ?? renvoie la première valeur *définie*.

```
let height = 0;

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



Comparaison
== vs ===

== vs ===

Comparaison

== 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.

https://developer.mozilla.org/fr/docs/Web/JavaScript/Equality_comparisons_and_sameness

• 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)

```
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
```

```
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 ==

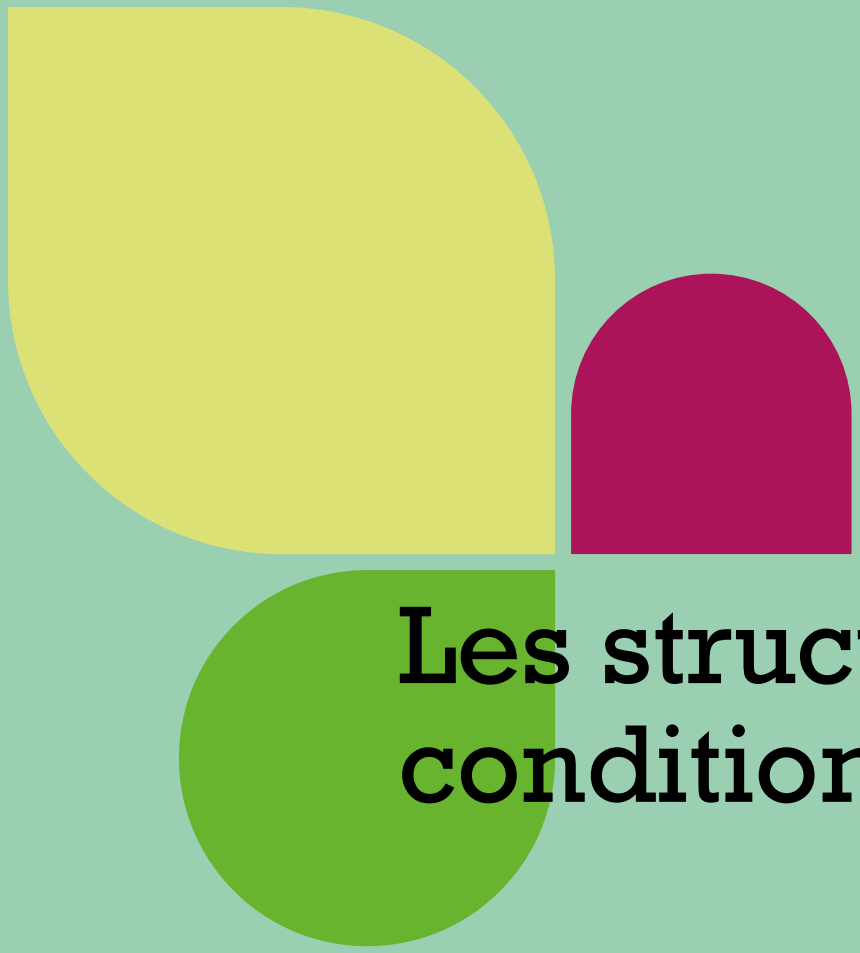
Conversions

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.

Tableau des conversions

		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 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

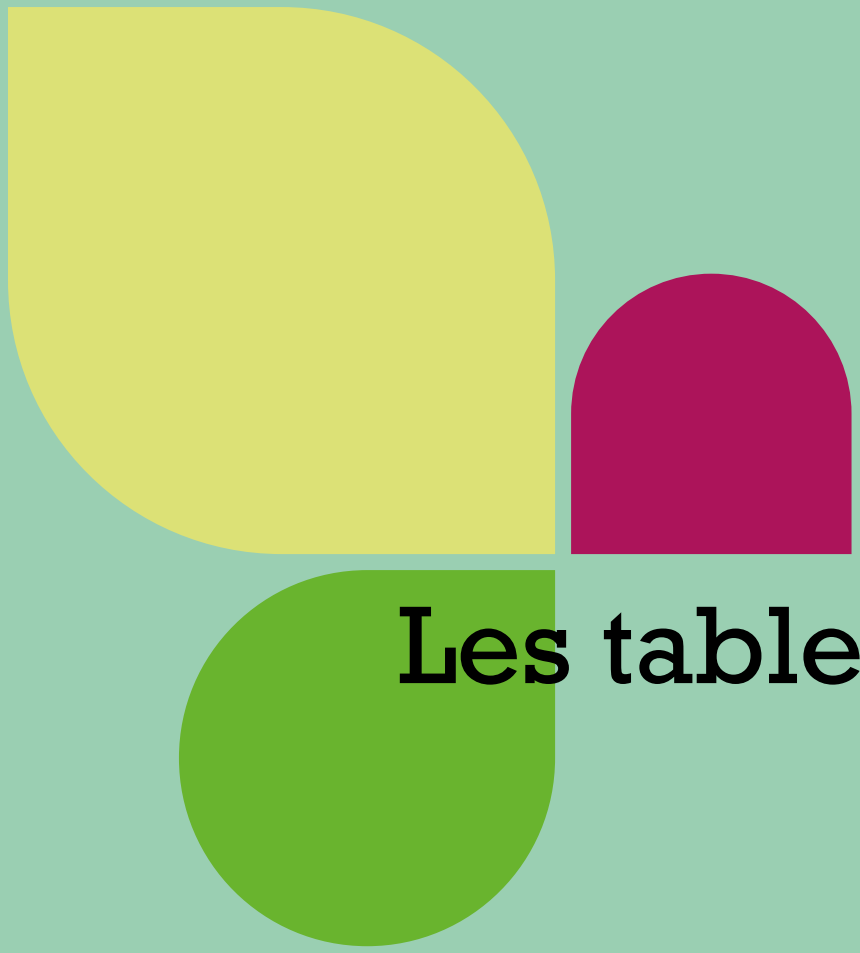
```
// Les répétitions
for (var i=0; i<100; i++) {
    console.log("Prèfère la boucle for si tu connais le nombre !");
}

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 tableaux

Les tableaux

En Javascript, nous avons la possibilité de créer des tableaux

https://developer.mozilla.org/fr/docs/Learn_web_development/Core/Scripting/Arrays

création

let tab=[valeur1, …, valeurN];
let tab=new Array();

Indice de départ : 0

recherche find()

suppression pop()

Voir Glossaire

ajout push()

Parcourir tab.forEach(num -> console.log(num));

Afpa

se former, avancer

Glossaire

Méthodes utiles à retenir pour les tableaux

AFPA CENTRE DE POMPEY

Les tableaux

objets très puissants avec plusieurs méthodes pratiques pour manipuler et travailler avec leurs éléments

push()

Ajoute un ou plusieurs éléments à la fin d'un tableau et retourne la nouvelle longueur du tableau.
let arr = [1, 2]; arr.push(3); ***arr devient [1, 2, 3]***

pop()

Supprime le dernier élément d'un tableau et le retourne.
let arr = [1, 2, 3]; arr.pop(); ***arr devient [1, 2] et donne 3***

shift()

Supprime le premier élément d'un tableau et le retourne.
let arr = [1, 2, 3]; arr.shift(); ***arr devient [2, 3] et donne 1***

unshift()

Ajoute un ou plusieurs éléments au début d'un tableau et retourne la nouvelle longueur du tableau.
let arr = [2, 3]; arr.unshift(1); ***arr devient [1, 2, 3]***

concat()

Fusionne 2 ou + tableaux et retourne un nouveau tableau.
let arr1 = [1, 2]; let arr2 = [3, 4];
let arr3 = arr1.concat(arr2); ***arr3 devient [1, 2, 3, 4]***

Les tableaux

objets très puissants avec plusieurs méthodes pratiques pour manipuler et travailler avec leurs éléments

join()

Retourne une chaîne de caractères formée par la concaténation des éléments du tableau, séparés par un délimiteur spécifié.
let arr = ['a', 'b', 'c']; let str = arr.join(', '); ***"a, b, c"***

slice()

Retourne une copie superficielle d'une portion du tableau dans un nouveau tableau.
let arr = [1, 2, 3, 4]; let sliced = arr.slice(1, 3); ***[2, 3]***

splice()

Permet de modifier un tableau en ajoutant, supprimant ou remplaçant des éléments à partir d'une position donnée.
let arr = [1, 2, 3, 4]; arr.splice(2, 1, 5);
arr devient [1, 2, 5, 4] (remplace 3 par 5)

forEach()

Exécute une fonction donnée sur chaque élément du tableau.
let arr = [1, 2, 3]; arr.forEach(num => console.log(num));
Affiche 1, 2, 3

map()

Crée un nouveau tableau avec les résultats de l'appel d'une fonction sur chaque élément du tableau d'origine.
*let arr = [1, 2, 3]; let doubled = arr.map(num => num * 2);*
[2, 4, 6]

Les tableaux

objets très puissants avec plusieurs méthodes pratiques pour manipuler et travailler avec leurs éléments

filter()

Crée un nouveau tableau contenant uniquement les éléments qui passent un test (défini par une fonction).
let arr = [1, 2, 3, 4]; let evenNumbers = arr.filter(num => num % 2 === 0); **donne *[2, 4]***

reduce()

Applique une fonction sur chaque élément du tableau (de gauche à droite) et retourne une seule valeur qui est le résultat de la réduction.
let arr = [1, 2, 3]; let sum = arr.reduce((acc, num) => acc + num, 0); **donne 6**

find()

Retourne le premier élément du tableau qui satisfait une condition donnée. let arr = [1, 2, 3, 4];
let found = arr.find(num => num > 2); ***donne 3***

indexOf()

Retourne l'index du premier élément trouvé qui correspond à la valeur donnée, ou -1 si l'élément n'existe pas

includes()

Vérifie si un élément est présent dans le tableau et retourne true ou false.

Les tableaux

objets très puissants avec plusieurs méthodes pratiques pour manipuler et travailler avec leurs éléments

sort()

Trie les éléments du tableau en place. (Peut être personnalisé avec une fonction de comparaison.)

reverse()

Inverse l'ordre des éléments dans le tableau.

some()

Vérifie si au moins un élément du tableau passe un test (renvoie true ou false). let arr = [1, 2, 3];
let hasEven = arr.some(num => num % 2 === 0); // ***true***

every()

Vérifie si tous les éléments du tableau passent un test.
let arr = [2, 4, 6]; let allEven = arr.every(num => num % 2 === 0); // ***true***

findIndex()

Retourne l'index du premier élément qui satisfait une condition. let arr = [1, 2, 3, 4];
let index = arr.findIndex(num => num > 2); // ***2***

Afpa

se former, avancer

Glossaire

Méthodes utiles de la classe Math

AFPA CENTRE DE POMPEY

Classe Math

Elle offre plusieurs méthodes très utiles pour effectuer des calculs mathématiques.

abs(x)	Retourne la valeur absolue de x (c'est-à-dire la distance de x à zéro, indépendamment de son signe).
round(x)	Retourne l'entier le plus proche de x. Si x est à égalité avec deux entiers, le plus proche vers zéro est retourné.
floor(x)	Retourne l'entier inférieur à x (arrondi à l'entier le plus bas).
ceil(x)	Retourne l'entier supérieur à x (arrondi à l'entier le plus haut).
max(x, y, ...)	Retourne le plus grand des nombres passés en arguments. Ça marche pour les tableaux : Math.max(...arr); ... opérateur spread

Classe Math

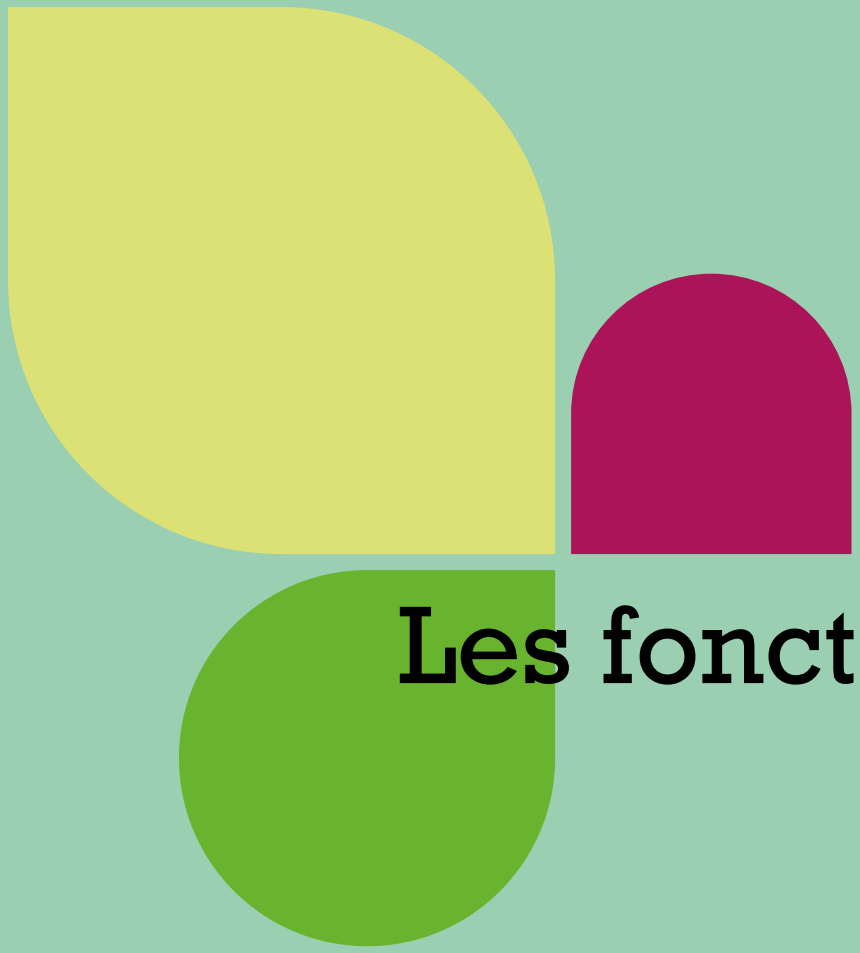
Elle offre plusieurs méthodes très utiles pour effectuer des calculs mathématiques.

min(x, y, ...)	Retourne le plus petit des nombres passés en arguments.
random()	Retourne un nombre pseudo-aléatoire entre 0 (inclus) et 1 (exclus).
pow(base, exponent)	Retourne base élevée à la puissance de exponent.
sqrt(x)	Retourne la racine carrée de x.
sin(x) cos(x) tan(x)	Ces méthodes retournent respectivement le sinus, le cosinus et la tangente de l'angle x exprimé en radians.

Classe Math

Elle offre plusieurs méthodes très utiles pour effectuer des calculs mathématiques.

log(x)	Retourne le logarithme naturel (base e) de x. Si vous voulez un logarithme à une autre base, vous pouvez utiliser une formule comme Math.log(x) / Math.log(base) pour la base que vous souhaitez.
exp(x)	Retourne e élevé à la puissance de x (soit Math.E^x).
trunc(x)	Retourne la partie entière de x en retirant sa partie décimale.
PI	Représente la constante π (pi) en JavaScript.
E	Représente la constante e (la base des logarithmes naturels).



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 fonctions callback, auto-exectucables, 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.

```
// 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.

```
// 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').

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.

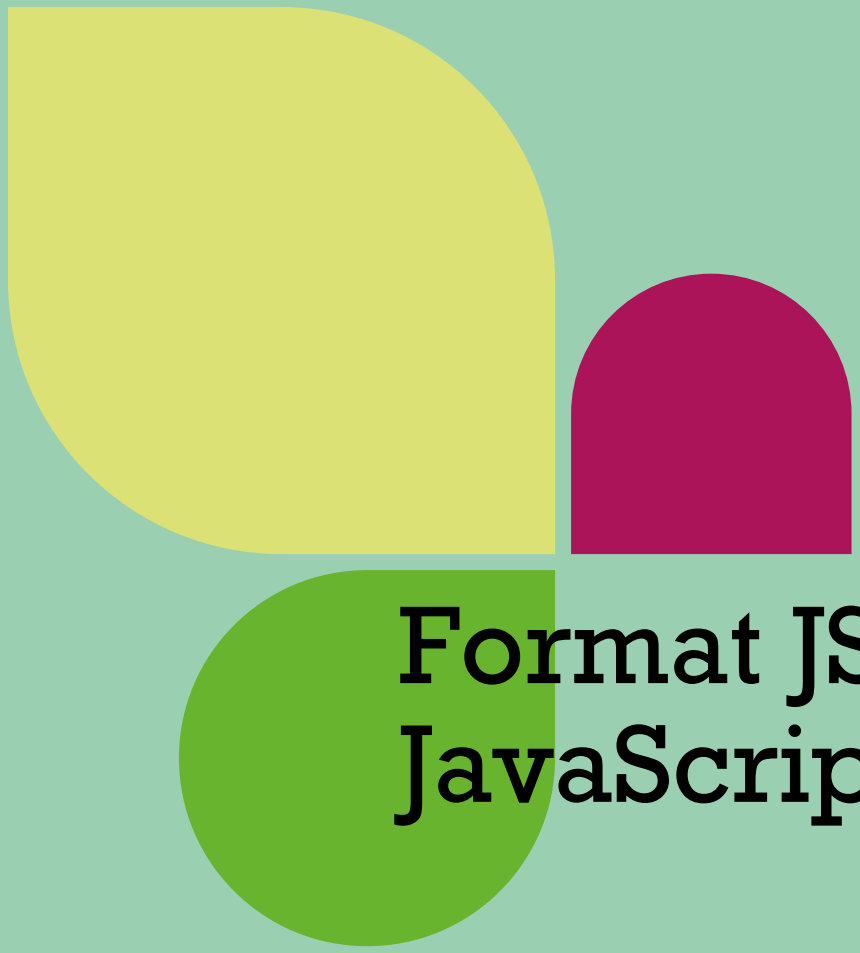
```
let myBook = {
    title: 'The Story of Tau',
    author: 'Will Alexander',
    numberOfPages: 250,
    isAvailable: true
};
```

Accès

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

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' }
```



Format JSON
JavaScript Object Notation

JSON

Format du web

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.

Pourquoi l'utiliser ?

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.

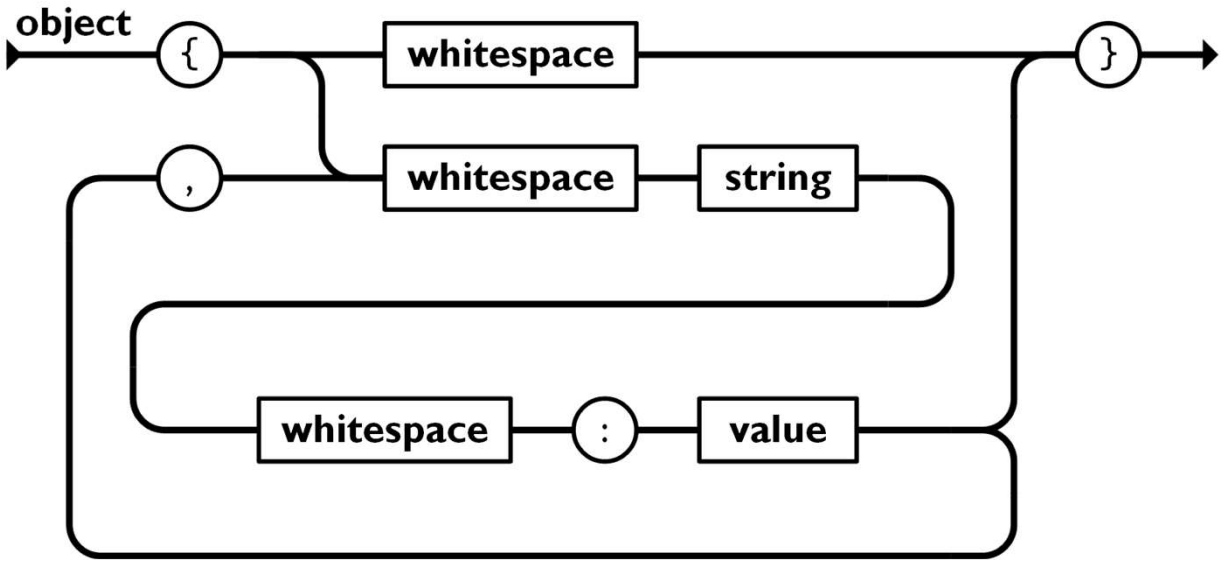
Définition

la syntaxe Json

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 différents types possibles

Les chaînes de caractères :

- { "firstName" : "Tom" }

Nombre

- { “age” : 30 }

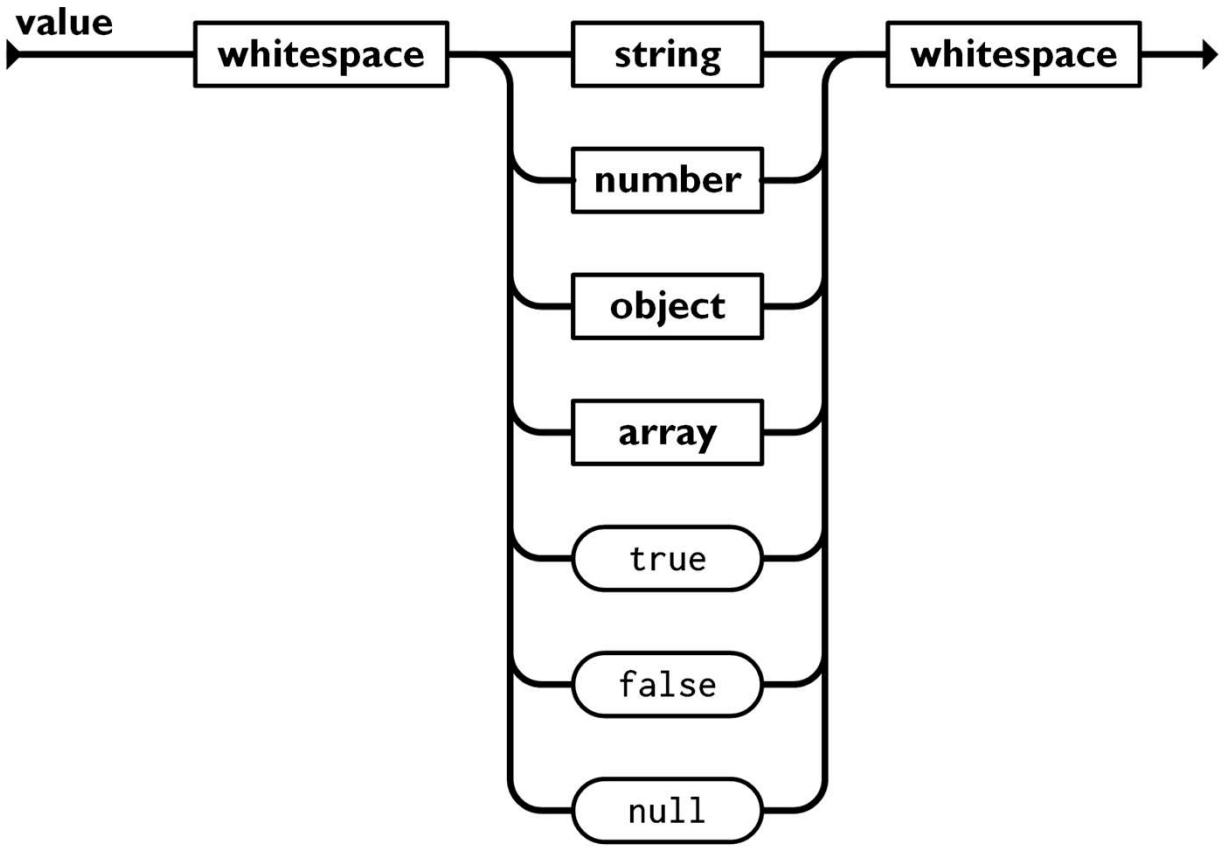
Booléen

- { “married” : false }

Null : Null est une valeur vide.

- { "bloodType" : null }

Les différents types



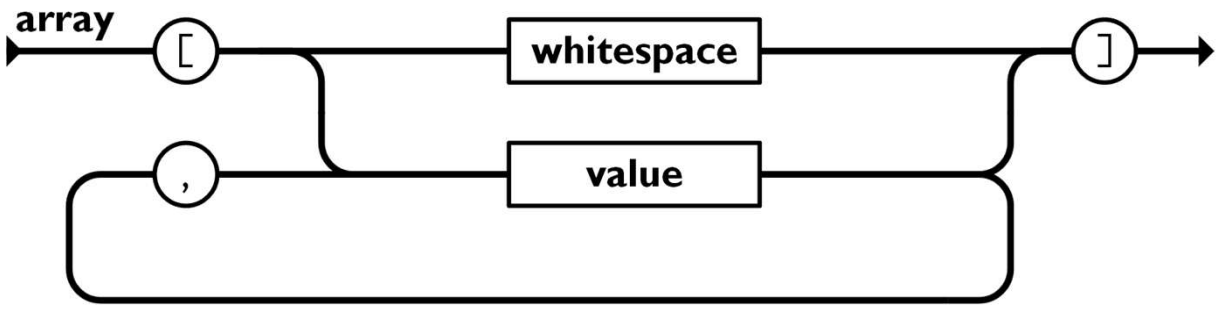
Les tableaux

Utilisation

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.

déclaration




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

Les objets JSON

Les objets JSON sont constitués de paires de deux composants constitués d'une clé et de sa valeur

- 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.

Les différents types

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

- Utiliser les tableaux :

Les valeurs sont placées entre crochets, et des virgules séparent chaque ligne.

Chaque valeur des tableaux JSON peut être d'un type différent.

```
{
    "firstName" : "Tom",
    "lastName" : "Jackson",
    "gender": "male",
    "hobby": [
        "football",
        "reading",
        "swimming"
    ]
}
```

Exemple

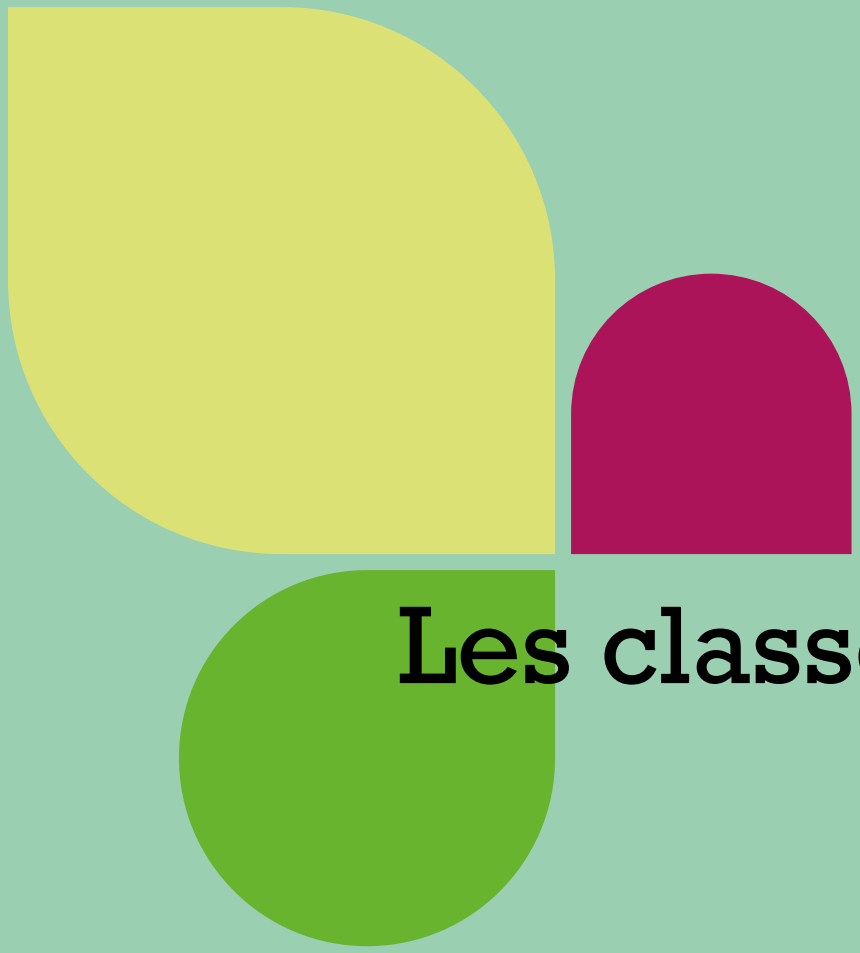
Un objet JSON

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, un tableau de membres.

Un petit exemple

```
{
    "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"}
        ]
}
```



Les classes en Javascript

Classe en ES6

Classe d'objets

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.

Création d'une classe

```
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

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

Static, constructeur, getter et setter

static

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
```

constructeur

getter

setter

```
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

Utilisation d'une classe dans un script

Supposons que vous avez une classe Person dans un fichier nommé Person.js. Il faut la déclarer en export

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

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

Importation

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"
```

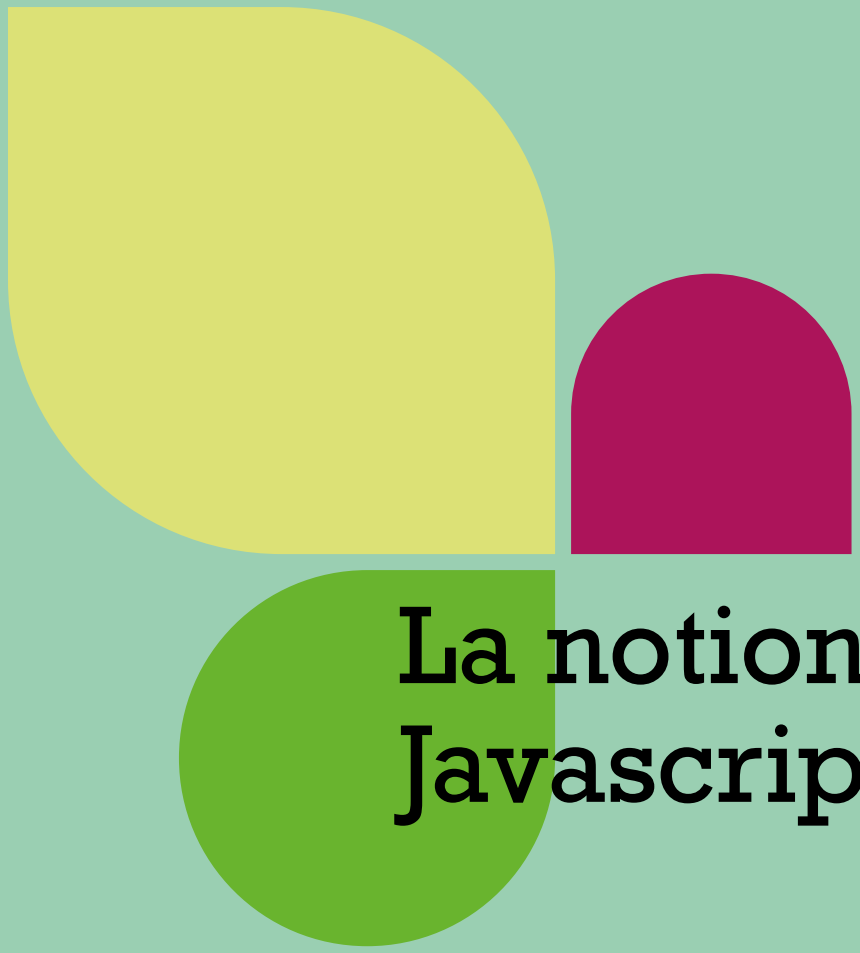
Dans ce cas, lors de la déclaration du script dans HTML, il faut le déclarer en module :

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

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.firstName + " " + this.lastName;
    }
}
```

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

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



La notion de prototype en Javascript

Les Prototypes

En JavaScript, tout est **objet** (sauf les types primitifs comme number, string, etc.).

Contrairement à d'autres langages (comme Java ou C++), JavaScript n'utilise pas de **classes** pour l'héritage, mais un mécanisme appelé prototype.

- C'est ce qu'on appelle l'héritage prototypique et diffère de l'héritage de classe (Java par exemple).

Qu'est-ce qu'un prototype ?

- Chaque objet en JavaScript a une propriété cachée appelée __proto__ (ou accessible via Object.getPrototypeOf()).
- Cette propriété pointe vers un autre objet : son prototype.
- Le prototype est un objet qui contient des propriétés et méthodes partagées par tous les objets qui en héritent.

```
const animal = { mange: true };
const chat = { miaule: true };
chat.__proto__ = animal; // chat hérite de animal

console.log(chat.mange); // true (hérité)
console.log(chat.miaule); // true (propre)
```

La chaîne de prototypes

Si une propriété/méthode n'est pas trouvée dans un objet, JavaScript la cherche dans son prototype, puis dans le prototype du prototype, etc.

Cela forme une chaîne de prototypes.

- La chaîne se termine à Object.prototype (le prototype par défaut de tous les objets), dont le prototype est null.

```
console.log(chat.__proto__ === animal); // true
console.log(animal.__proto__ === Object.prototype); // true
console.log(Object.prototype.__proto__); // null
```

Constructeurs et prototypes

- Les fonctions constructeurs (utilisées avec new) ont une propriété prototype.
- Quand un objet est créé avec new, son __proto__ pointe vers constructeur.prototype.

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]"
```