

Votre Formation Sur Mesure



Javascript

ELAN

14 rue du Rhône - 67100 STRASBOURG

☎ 03 88 30 78 30 📧 elan@elan-formation.fr

www.elan-formation.fr

SAS ELAN au capital de 37 000 € -

RCS Strasbourg B 390758241 – SIRET 39075824100041 – Code APE : 8559A

N° déclaration DRTEFP 42670182967 - Cet enregistrement ne vaut pas agrément de l'État

TABLE DES MATIERES

I.	Introduction	4
II.	Création d'un projet web simple « Hello World »	5
1.	Création du dossier racine	5
2.	Création du fichier index.html.....	6
3.	Création du fichier JS	7
4.	Import du fichier JS dans le fichier HTML.....	8
5.	Premier test d'exécution dans le navigateur.....	8
III.	Les commentaires	10
IV.	Instruction ou bloc	11
1.	Les instructions.....	11
2.	Les blocs.....	11
V.	Les variables	13
1.	Utilités des variables.....	13
2.	Règles de bonnes pratiques	13
3.	Les types de variables les plus utiles	13
a.	La variable « let »	13
b.	La constante « const ».....	14
4.	Les types de données primitifs les plus utiles	14
a.	number	14
b.	string.....	14
c.	boolean.....	15
5.	La portée des variables	15
VI.	Les opérateurs	16
1.	Affectations et calculs.....	16
a.	L'opérateur d'affectation simple.....	16
b.	Les opérateurs arithmétiques	16
c.	Les opérateurs d'affectation spéciaux	17
d.	L'opérateur de concaténation.....	17
2.	Comparaisons.....	18
a.	Les opérateurs d'égalité et d'inégalité.....	18
b.	Les opérateurs de comparaison	18
3.	Conditions	19
a.	Les opérateurs logiques	19
b.	L'opérateur ternaire	19

VII. Les blocs les plus utiles	20
1. Les blocs de condition(s)	20
a. Le « if » ou « if/elseif/else »	20
b. Le « switch/case »	23
2. Les boucles les plus utiles	24
a. La boucle « for »	24
b. La boucle « while »	27
VIII. Les fonctions	28
1. Utilités des fonctions	28
2. Exemples	29
IX. Les tableaux.....	32
1. Les tableaux indexés.....	32
a. Utilités	32
b. Les bases de leur utilisation	33
c. Quelques méthodes utiles	35
2. Les tableaux associatifs.....	37
a. Utilités	37
b. Les bases de leur utilisation	37
3. Exemples d'application	40
a. Tableaux indexés et associatifs avec Pikachu	40
X. DOM.....	41
1. Définition	41
2. En pratique.....	41
a. Atteindre des éléments du DOM	42
b. Modifier des éléments du DOM.....	44

I. Introduction

Javascript, aussi appelé **JS**, est le langage qui a rendu possible le « **dynamisme côté client** » de toutes les applications web.

Depuis, grâce à Node.js, il peut également être utilisé côté serveur, ce qui en fait le seul langage pouvant être utilisé en frontend et en backend.

Dans ce document, nous allons aborder Javascript de la façon la plus **simple** et nous concentrer sur **les notions les plus importantes et utiles**.

Nous allons débiter par la création étape par étape d'un projet web simple permettant d'utiliser JS, puis parcourir les **fondamentaux**.

II. Création d'un projet web simple « Hello World »

Dans cette partie, nous allons créer un nouveau projet web simple, afin de pouvoir exécuter du code Javascript.

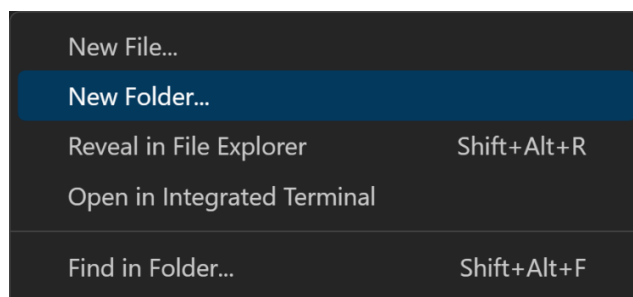
Vous pourrez reprendre cette façon de faire pour vos futurs projets JS.

1. Création du dossier racine

Le **dossier racine** d'un projet est le **dossier principal**, celui qui contient tous les autres dossiers et tous les fichiers.

Le nom du dossier racine est le nom du projet.

Dans VSCode -> clic droit sur un dossier -> New Folder... :



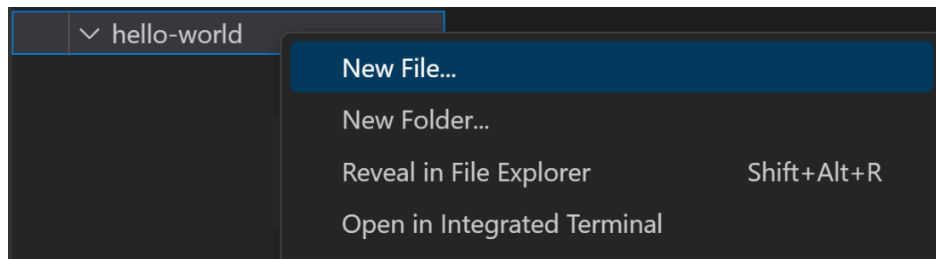
Lui donner le nom du projet :

A screenshot of a text input field in VS Code. The field contains the text 'hello-world' and has a blue border. To the left of the field is a greater-than sign '>'. The background is dark, matching the VS Code theme.

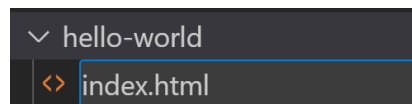
2. Création du fichier index.html

Le fichier index.html doit se trouver à la racine du projet.

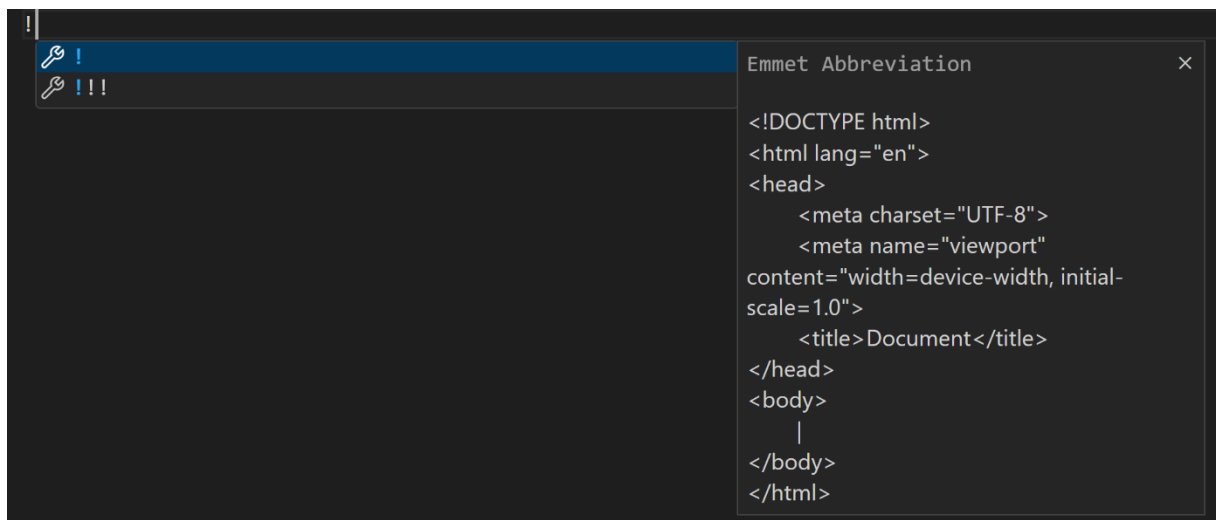
Clic droit sur le dossier racine (hello-world) -> New File... :



Lui donner le nom « index.html » :



Utiliser Emmet (plugin intégré dans VSCode) afin d'obtenir le template HTML5 de base grâce au symbole « ! » :



Ajouter un petit contenu afin de ne pas avoir une page blanche lors du premier test (par exemple un titre <h1> dans le <body>).

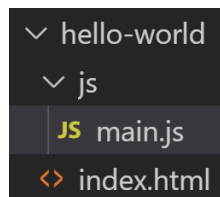
index.html contient maintenant le code suivant :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Hello JS</title>
  </head>
  <body>
    <h1>Hello JS</h1>
  </body>
</html>
```

3. Création du fichier JS

Créer un dossier « js » dans le dossier racine, puis créer un fichier « main.js » dans ce nouveau dossier « js ».

Nous avons maintenant la structure de base pour ce petit projet :



```

  ✓ hello-world
    ✓ js
      JS main.js
      <> index.html
```

La façon la plus simple de faire s'exprimer Javascript est d'afficher un message dans la **console**. Tous les navigateurs en ont une.

Dans le fichier main.js, ajouter cette ligne de code, qui va écrire « Hello JS! » dans la console :

```
console.log("Hello JS!");
```

4. Import du fichier JS dans le fichier HTML

Ajouter une balise <script> juste avant la fin du <body>.

Lui mettre comme valeur d'attribut HTML « src » le chemin du fichier JS :

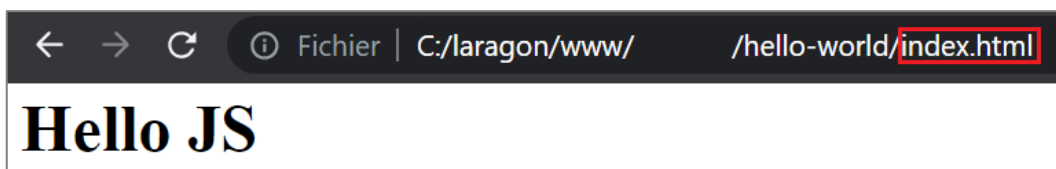
```
<body>
  <h1>Hello JS</h1>

  <script src="js/main.js"></script>
</body>
```

5. Premier test d'exécution dans le navigateur

Nous devons tester ce petit projet, afin de valider que tout fonctionne.

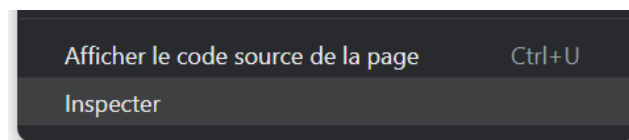
Ouvrir le fichier index.html dans un navigateur (j'utilise ici Google Chrome) :



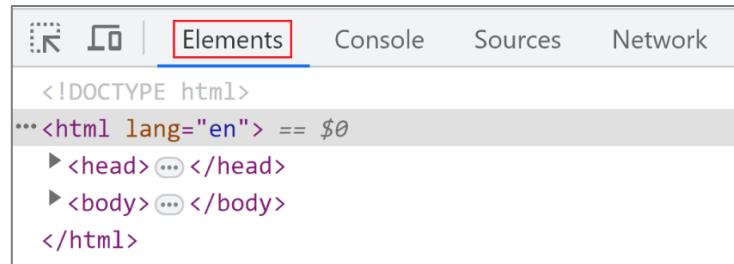
Observer que le titre <h1> « Hello JS » s'affiche bien dans la page. Cela valide la partie HTML.

Ensuite il faut ouvrir la console du navigateur, afin de valider que la partie Javascript fonctionne également.

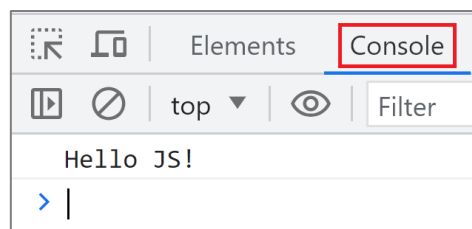
Clic droit dans la page du navigateur -> Inspecter (peut être légèrement différent en fonction du navigateur) :



L'inspecteur s'ouvre par défaut sur l'onglet « Eléments », qui permet d'observer le HTML et le CSS :



Aller dans l'onglet « Console », qui permet d'ouvrir la console et d'observer le JS :



Observer que le message « Hello JS! » a bien été écrit dans la console. Cela valide la partie JS.

III. Les commentaires

Les **commentaires** sont utiles pour organiser son code, laisser des **explications** sur la logique appliquée, prendre des **notes**, etc.

Ils ne sont pas interprétés, ils n'auront donc **aucun effet lors de l'exécution** du programme.

Ils prennent une **couleur verte** dans VSCode.

Il y a 2 types de commentaires basiques en JS :

- « **//** » : à partir de « **//** » et jusqu'à la fin de la ligne, ce qui est écrit est un commentaire
- « **/* */** » : à partir de « **/*** » et jusqu'à « ***/** », ce qui est écrit est un commentaire (sur une ou plusieurs lignes)

```
// log de Hello JS!  
console.log("Hello JS!"); // dans la console  
  
/*  
    Apprenons à coder  
    en Javascript  
*/
```

Raccourci clavier : « **ctrl + :** » pour commenter une ligne sur VScode.

IV. Instruction ou bloc

Chaque sujet de code JS dans la suite de ce document sera soit une instruction, soit un bloc.

Nous allons parcourir différentes instructions et différents blocs, mais il est utile de savoir à l'avance que ce sont **les deux grandes familles** dans l'écriture de code.

1. Les instructions

Une **instruction** est un **ordre direct**. Le programme doit effectuer une action.

Les lignes d'instruction(s) **se terminent par un « ; »**.

Syntaxe globale :

```
instruction1;  
instruction2;
```

Exemples concrets simples :

```
console.log("Hello JS!");  
let age = 37;
```

2. Les blocs

Un **bloc** est une **structure**. Il permet de faire **appliquer une logique** au programme.

Un bloc contient un **« corps »**, qui va de **« { » à « } »**, dans lequel on peut placer une/des instruction(s) et/ou bloc(s).

Les blocs ont un nom et une syntaxe qui répond à leurs besoins, ils sont paramétrables.

Syntaxe globale de la plupart des blocs :

```
nomBloc (parametrage) {  
    // instruction(s) et/ou bloc(s)  
}
```

Exemples concrets simples :

```
if (age >= ageMin) {  
  
}  
  
while (k < 10) {  
  
}
```

V. Les variables

1. Utilités des variables

Les variables servent à **stocker des données**.

Elles ont un **type** (nombre, chaîne de caractères, etc.).

Elles peuvent être **utilisées** dans le code, **une ou plusieurs fois**.

Le fait d'être **réutilisables** est très important, afin de ne pas refaire les mêmes calculs plusieurs fois.

Elles peuvent également aider à écrire du **code plus lisible**, en **décomposant** une suite d'opérations simples à la place d'un calcul unique très complexe.

2. Règles de bonnes pratiques

Une **variable** porte un **nom écrit en « camelCase »** : les lettres sont en minuscule, sauf la 1^{ère} lettre de chaque mot à partir du 2^{ème} mot qui est en majuscule.

Le nom d'une variable ne doit pas contenir d'accent, d'espace, de « _ » ni aucun autre symbole spécial, seulement des lettres sans accent en camelCase et éventuellement des chiffres (mais pas en début de nom).

Exemples :

```
nom  
deuxMots  
unPeuPlusDeMots  
longNomDeVariableEnCamelCase  
eteEnsoleille // sans accent, pas "étéEnsoleillé"  
segmentPoint1
```

3. Les types de variables les plus utiles

a. La variable « let »

Une variable classique est **déclarée grâce au mot-clef « let »**.

Sa valeur **peut ensuite être modifiée** autant de fois qu'on le souhaite.

Exemple :

```
let animal = "dog"; // déclaration et initialisation de la variable animal
animal = "cat"; // modification de sa valeur
animal = "bird"; // modification de sa valeur
```

b. La constante « const »

Une constante est une variable dont **la valeur ne peut pas être modifiée**.

Tenter de modifier la valeur d'une constante provoque une erreur.

Exemple :

```
const deux = 2; // la valeur ne peut pas être modifiée
// la ligne suivante est mise en commentaire car elle provoque une erreur
// deux = 3; // -> Uncaught TypeError: Assignment to constant variable.
```

4. Les types de données primitifs les plus utiles

a. number

Un « **number** » est un **nombre, entier ou décimal**.

Exemples :

```
const deux = 2;
let quantite = 3;
const prix = 1.15;
```

b. string

Un « **string** » est une **chaîne de caractères**.

Lorsqu'on veut écrire sa valeur, on le fait entre **guillemets / quotes**.

Exemples :

```
const prenom = "John";
let repas = "pizza";
const criDuChien = "abolement";
```

c. boolean

Un « **boolean** » est un **booléen**, qui peut avoir comme valeur « **true** » (**vrai**) ou « **false** » (**faux**).

Par convention, le nom d'une variable de type « boolean » **commence** par « **is** » (**est**) ou « **has** » (**a**), afin d'être identifiée facilement.

Exemples :

```
let isAdult = true;
let isTaskDone = false;
const hasBeenToTokyo = true;
```

5. La portée des variables

Une **variable** a une certaine « **portée** ». Il s'agit de la **zone de code** dans laquelle elle peut être utilisée, dans laquelle elle est **définie**.

Cette portée s'étend :

- depuis la ligne à laquelle la variable a été déclarée (grâce à « **let** » ou « **const** »)
- jusqu'à la fin du bloc (« **}** ») dans lequel la variable a été déclarée

Note : elle sera accessible dans les blocs intérieurs.

Exemple imagé de structure :

```
{
  // début du bloc dans lequel age est déclarée
  let age = 37; // début de sa portée
  {
  }
} // fin du bloc dans lequel age est déclarée = fin de sa portée
}
```

VI. Les opérateurs

Les **opérateurs** sont des **symboles** qui ont chacun un **effet précis**.

Il en existe beaucoup, nous allons découvrir ceux qui sont les plus **utiles**.

1. Affectations et calculs

a. L'opérateur d'affectation simple

L'opérateur d'**affectation** « = » permet d'**attribuer une valeur à une variable**.

Nous en avons déjà vu plusieurs exemples :

```
let price = 1.17;  
const animal = "dog";  
let isAdult = true;
```

b. Les opérateurs arithmétiques

Ces opérateurs demandent d'être **entourés de valeurs de type « number »**, puisqu'ils entraînent un calcul mathématique. Ils **renvoient un « number »**.

Nous avons accès aux opérateurs arithmétiques basiques :

■ « + » : addition

■ « - » : soustraction

■ « * » : multiplication

■ « / » : division

Nous y ajouterons l'opérateur « **modulo** », qui s'utilise avec le symbole « % ».

Le **modulo**, pour l'expression « **a % b** » (qui **se prononce « a modulo b »**), est le reste de la division entière de a par b.

Par exemple :

■ « $7 \% 3$ » vaut 1, car $7/3$ vaut 2 en division entière, il reste donc 1 [$7 - (2 * 3)$]

$$\Rightarrow 7 = 2 * 3 + 1$$

■ « $6 \% 2$ » vaut 0, car $6/2$ vaut 3 en division entière, il reste donc 0 [$6 - (3 * 2)$]

$$\Rightarrow 6 = 3 * 2 + 0$$

■ « $2 \% 9$ » vaut 2, car $2/9$ vaut 0 en division entière, il reste donc 2 [$2 - (0 * 9)$]

$$\Rightarrow 2 = 0 * 9 + 2$$

c. Les opérateurs d'affectation spéciaux

Il existe certains opérateurs qui **cumulent plusieurs actions**. Voici les opérateurs les plus utiles qui entraînent **un calcul puis une affectation** :

■ « $+=$ » : addition puis affectation

■ « $-=$ » : soustraction puis affectation

■ « $*=$ » : multiplication puis affectation

■ « $/=$ » : division puis affectation

■ « $\%=$ » : modulo puis affectation

d. L'opérateur de concaténation

Concaténer, c'est le fait de **fusionner plusieurs chaînes de caractères** ensemble, afin d'en obtenir une seule.

Cet opérateur **renvoie un « string »**.

Son symbole est « $+$ » en JS, comme pour l'addition. Si ce symbole n'est pas entouré de deux valeurs de type « number », alors le programme le verra comme une concaténation et non pas une addition.

2. Comparaisons

a. Les opérateurs d'égalité et d'inégalité

Voici les opérateurs qui testent une **égalité** ou une **inégalité** entre les **valeurs** qui les entourent :

- « == » : même valeur
- « === » : même valeur et même type
- « != » : valeur différente
- « !== » : valeur différente ou type différent

Ils renvoient un « **boolean** ».

b. Les opérateurs de comparaison

Voici les opérateurs qui comparent les valeurs qui les entourent :

- « < » : inférieur strictement
- « <= » : inférieur ou égal
- « > » : supérieur strictement
- « >= » : supérieur ou égal

Ils renvoient un « **boolean** ».

3. Conditions

a. Les opérateurs logiques

Un opérateur logique sert principalement à **lier plusieurs conditions**, souvent des « boolean », afin de savoir si l'**ensemble** est « true » (vrai) ou « false » (faux).

Tous les opérateurs logiques **renvoient un « boolean »**. Voici les plus utiles :

- « **&&** » : le « **et** » logique, qui sert à savoir **si deux conditions sont vraies**
- « **||** » : le « **ou** » logique, qui sert à savoir **si au moins une des deux conditions est vraie**
- « **!** » : le « **non** » logique, qui sert à **inverser une condition**

b. L'opérateur ternaire

L'opérateur ternaire s'écrit en **3 parties**, Il est donc composé de **2 symboles** qui séparent ces 3 parties : « **?** » et « **:** ».

Il teste une **condition**, puis retourne quelque chose si la condition est **vraie** ou autre chose si elle est **fausse**.

Syntaxe globale : **condition ? siVrai : siFaux**

- « condition » : la **condition testée**, généralement un « boolean »
- « siVrai » : ce qui est **retourné** par l'opérateur **si la condition est vraie**
- « siFaux » : ce qui est **retourné** par l'opérateur **si la condition est fausse**

VII. Les blocs les plus utiles

1. Les blocs de condition(s)

a. Le « if » ou « if/elseif/else »

Le point commun des variantes du bloc « if », c'est que le programme exécutera 0 ou 1 bloc, mais jamais plusieurs.

➤ Le « if » simple (si quelque chose est vrai)

Si la **condition** du « if » est « **true** » (vraie), le programme **entrera dans son bloc**, c'est-à-dire qu'il exécutera le contenu du corps du bloc (qui va de « { » à « } »).

Si la **condition** est « **false** » (fausse), il **n'entrera pas dans le bloc**, il le « **sautera** » et continuera sa lecture du code après la fermeture du bloc (« } »).

Exemple (si la variable age est supérieure ou égale à 18, alors on écrit le message dans la console [sinon on ne fait rien]) :

```
// le bloc if simple (si la condition est vraie)
if (age >= 18) {
    console.log("La personne est majeure.");
}
```

➤ Le « if/else » (si quelque chose est vrai, sinon)

Si la **condition** du « if » est « **true** » (vraie), le programme **entrera dans le bloc du « if »**.

Sinon, donc si la **condition** du « if » est « **false** » (fausse), le programme **entrera dans le bloc du « else »**.

Remarque : le **bloc « else » n'a pas de condition**, il représente le « **sinon** », il est exécuté si la condition du « if » est fausse (fausse).

Exemple (si la variable age est supérieure ou égale à 18, alors on écrit un message dans la console, sinon on écrit l'autre message) :

```
// la variante if/else (si la condition est vraie, sinon)
if (age >= 18) {
    console.log("La personne est majeure.");
} else {
    console.log("La personne est mineure.");
}
```

- Le « **if/elseif** » (si quelque chose, sinon si autre chose, sinon si...)

Si la condition du « if » est « true » (vraie), le programme entrera dans le bloc du « if ».

Sinon, si la condition du premier « elseif » est « true » (vraie), le programme entrera dans son bloc à la place.

Sinon, si la condition du deuxième « elseif » est « true » (vraie), le programme entrera dans son bloc à la place.

Etc., dans l'ordre d'écriture. **L'ordre des conditions est important.**

Remarques :

- quand le programme rencontre la première condition « true » (vraie), il entre dans le bloc correspondant, exécute le code de ce bloc, puis va directement à la fin du groupe « if/elseif » entier
- le programme n'exécutera pas plusieurs blocs, mais uniquement le premier dont la condition est « true »
- si les conditions du « if » et des « elseif » ne couvrent pas 100% des cas, il est possible qu'aucun bloc ne soit exécuté

Exemple (si la personne a au moins 100 ans, sinon si elle a au moins 80 ans, sinon si... [attention on ne fait rien ici si une personne a moins de 40 ans]) :

```
// la variante if/elseif (si une condition est vraie, sinon si une autre condition...)
if (age >= 100) {
    console.log("La personne est centenaire.");
} else if (age >= 80) {
    console.log("La personne est au moins octogénaire.");
} else if (age >= 60) {
    console.log("La personne est au moins sexagénaire.");
} else if (age >= 40) {
    console.log("La personne est au moins quadragénaire.");
}
```

- Le « **if/elseif/else** » (si quelque chose, sinon si autre chose, sinon si..., sinon)

Cette forme reprend les mêmes notions et comportements que pour la forme « if/elseif », mais on y ajoute un « else » (sinon).

Puisqu'il y a un « else », le programme exécutera forcément 1 et 1 seul bloc :

- soit le premier bloc « if » ou « elseif » rencontré qui aura sa condition « true »
- soit par défaut le bloc « else » qui couvre le reste des possibilités (s'il est exécuté, cela signifie que toutes les conditions du « if » et des « elseif » sont « false »)

Exemple (si la personne a au moins 100 ans, sinon si elle a au moins 80 ans, sinon si..., sinon) :

```
// la variante complète if/elseif/else (si la condition est vraie, sinon si une autre condition, sinon si..., sinon)
if (age >= 100) {
    console.log("La personne est centenaire.");
} else if (age >= 80) {
    console.log("La personne est au moins octogénaire.");
} else if (age >= 60) {
    console.log("La personne est au moins sexagénaire.");
} else if (age >= 40) {
    console.log("La personne est au moins quadragénaire.");
} else {
    console.log("La personne n'est pas encore quadragénaire.");
}
```

Résumé sur les points importants des blocs de type « if/elseif/else » :

- le bloc « if » est toujours le premier bloc du groupe
- le bloc « else » est toujours le dernier bloc du groupe
- le bloc « else » est le seul qui n'ait pas de condition, car il représente le « sinon », toutes les autres possibilités
- si vous avez seulement besoin d'exécuter une partie de code si une condition est vraie, alors un bloc « if » simple suffit
- si vous avez besoin d'un bloc par défaut à exécuter si aucun autre ne l'est, alors il vous faut un bloc « else »
- si vous utilisez au moins un bloc « elseif », alors faites attention à l'ordre des conditions
- si vous n'avez pas de bloc « else », alors entre 0 et 1 bloc sera exécuté
- si vous avez un bloc « else », alors 1 et 1 seul bloc sera exécuté
- si vous avez plusieurs conditions à tester et que vous voulez laisser la possibilité d'entrer dans plusieurs blocs, alors vous aurez besoin de plusieurs blocs « if » séparés (voir les deux points précédents)

b. Le « switch/case »

Le bloc « **switch** » prend un argument en paramètre et va tester si l'égalité est « true » (vraie) entre ce paramètre et chaque « **case** », **dans l'ordre d'écriture** des « case ».

Dès qu'une égalité est trouvée, le programme entre dans le bloc « case » associé.

S'il **rencontre** le bloc « **default** », alors cela signifie qu'aucun « case » n'était égal au paramètre du « switch ». Dans ce cas, il **entre dans ce bloc par défaut**.

Si l'instruction « **break;** » (« casser » en français) est lue / exécutée à l'intérieur d'un bloc « switch/case », alors le programme « **saute** » le **bloc entier** et continue sa lecture après le « } » final du bloc « switch/case ».

Lorsqu'il **entre dans un bloc**, alors le programme exécute le code **depuis le « : »** associé au « case » (ou au « default ») et **jusqu'à rencontrer un « break » ou arriver à la fin du bloc « switch/case » entier**. Dans ce cas, les lignes « case ...: » sont ignorées / sautées.

Cas standard (chaque bloc « case » se termine par un « break » et il y a un bloc « default ») :

```
let action = "virement";

switch (action) {
  case "retrait":
    // instructions de retrait
    break;

  case "virement":
    // instructions de virement
    break;

  case "cloturerCompte":
    // instructions de cloture de compte
    break;

  default:
    // instructions par défaut
    break;
}
```

Cas sans « break » systématique (certaines instructions peuvent être exécutées en entrant par différents « case ») :

```
let numberToGuess = 7;
let operation = "foisDeuxPlusTrois";

switch (operation) {
  case "plusUn":
    numberToGuess++;
    break;

  case "foisDeuxPlusTrois":
    numberToGuess *= 2;

  case "plusTrois":
    numberToGuess += 3;
    break;

  case "foisZero":
  default:
    numberToGuess = 0;
    break;
}
```

2. Les boucles les plus utiles

Les **boucles** sont des **blocs** qui permettent d'**exécuter plusieurs fois** le code contenu dans leur corps.

Chaque passage dans une boucle est appelé une **itération**.

a. La boucle « for »

En général, on utilise la boucle for (ou une de ses variantes) quand **le programme sait combien d'itérations** il devra effectuer au moment où il arrive sur cette boucle.

On itère « n fois », « pour chaque... parmi... ».

Syntaxe : for (initialisation; condition pour continuer; incrémentation) {}

Voici quelques **exemples d'utilisation** :

Le « for » « pour chaque entier de a à b inclus » (ici pour i allant de 1 à 10 inclus, écrivant progressivement la table de multiplication des 3 dans la console) :

```
// le for "de a à b inclus"
for (let i = 1; i <= 10; i++) {
  console.log(`3 x ${i} = ${3 * i}`);
}
```



3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30

Le « for » « n fois » (ici « 3 fois », pour i allant de 0 inclus à 3 exclu [donc jusqu'à 2 inclus], écrivant 3 fois au total la phrase dans la console) :

```
// le for "n fois"
for (let i = 0; i < 3; i++) {
  console.log("Je serai un bon développeur");
}
```



3 Je serai un bon développeur

Le « for » « de a à b inclus avec un pas différent de 1 » (ici pour i démarrant à 3, incrémenté de 3 à chaque itération, tant que i <= 10) :

```
// le for "de a à b inclus avec un pas différent de 1"
for (let i = 3; i <= 10; i += 3) {
  console.log(`${i} est inférieur ou égal à 10 et est divisible par 3`);
}
```



3 est inférieur ou égal à 10 et est divisible par 3
6 est inférieur ou égal à 10 et est divisible par 3
9 est inférieur ou égal à 10 et est divisible par 3

Le « for » « inversé » (on décrémente au lieu d'incrémenter) (ici pour i démarrant à 5, décrémente de 1 à chaque itération, jusqu'à 0 exclu [donc 1 inclus]) :

```
// le for "inversé" (on décrémente au lieu d'incrémenter)
for (let i = 5; i > 0; i--) {
  console.log(`${i}...`);
}
console.log("Bonne année !");
```



5...
4...
3...
2...
1...
Bonne année !

b. La boucle « while »

En général, on utilise la boucle while (ou une de ses variantes) quand **le programme ne sait pas combien d'itérations** il devra effectuer au moment où il arrive sur cette boucle.

On itère « tant que... ».

Syntaxe : while (condition pour continuer) {}

Notes :

- **risque de boucle infinie** (rester indéfiniment bloqué dans le while et ne jamais en sortir) : si dans le corps du while rien ne fait que la condition pour continuer renverra false => risque de **crash de l'application**. Penser à double-checker que la condition renverra bien false au bout d'un moment

Exemple d'utilisation :

Le « while » classique (ici on écrit la phrase dans la console tant que la variable est strictement inférieure à 18) :

```
let kAge = 12;
while (kAge < 18) {
  console.log(`J'ai maintenant ${kAge} ans...`);
  kAge++;
}
console.log(`J'ai finalement ${kAge} ans.`);
```



J'ai maintenant 12 ans...
J'ai maintenant 13 ans...
J'ai maintenant 14 ans...
J'ai maintenant 15 ans...
J'ai maintenant 16 ans...
J'ai maintenant 17 ans...
J'ai finalement 18 ans.

VIII. Les fonctions

Afin de **ne pas dupliquer de code**, lorsqu'une **logique commune** doit être utilisée à plusieurs endroits, on crée une fonction.

Une fonction a un **nom** (écrit en camelCase), **des arguments** (paramètres) et un corps (bloc « {} ») qui contient l'implémentation d'une **logique** et **peut retourner un résultat**.

Une fois la fonction **déclarée**, elle peut être **appelée** autant de fois que nécessaire.

Lors de l'appel à une fonction, **l'ordre des arguments** doit être respecté.

Les **arguments** sont des **variables locales** et peuvent être utilisés à l'intérieur de la fonction.

Syntaxes globales d'une fonction (déclaration et appel) :

- Nom : **nomDeLaFonction**
- Arguments : oui, il y en a 3 -> **arg1, arg2, arg3**
- Implémentation : **calcul** de $\text{arg1} * \text{arg2} + \text{arg3}$
- Renvoi d'un résultat : oui -> le **résultat du calcul**

```
// déclaration de la fonction
function nomDeLaFonction(arg1, arg2, arg3) {
  // logique implémentée
  const result = arg1 * arg2 + arg3;
  // renvoi d'un résultat
  return result;
}

// appel à la fonction déclarée plus haut
const valeurRenvoyee = nomDeLaFonction(1, 2, 3);
```

1. Utilités des fonctions

Les fonctions sont très utiles car elles permettent de :

- réutiliser une logique implémentée
- éviter la duplication de code
- structurer et organiser le code par parties
- améliorer la lisibilité du code

2. Exemples

Une fonction simple (ici qui prend 1 argument et retourne le résultat de la multiplication par 3 de cet argument) :

```
// déclaration de la fonction getNombreFoisTrois, qui prend en argument
// un nombre et qui renvoie le triple de ce nombre
function getNombreFoisTrois(nombre) {
    // nombre est un argument et une variable locale
    return nombre * 3; // la fonction retourne le résultat du calcul
}
```

```
// appel à la fonction getNombreFoisTrois en lui passant 5 en argument
// et affectation du résultat reçu dans la constante cinqFoisTrois
const cinqFoisTrois = getNombreFoisTrois(5); // cinqFoisTrois = 15
```

```
// log de cinqFoisTrois dans la console
console.log("cinqFoisTrois = " + cinqFoisTrois);
```



cinqFoisTrois = 15

Une fonction sans résultat retourné (ici log de la phrase fournie en argument, à laquelle on ajoute un espace et un point d'exclamation) :

```
// une fonction sans résultat retourné
function logPhraseAvecPointDExclamation(phrase) {
    console.log(`${phrase} !`);
}
```

```
logPhraseAvecPointDExclamation("Je serai un bon développeur");
```



Je serai un bon développeur !

Une fonction sans argument (ici renvoie une valeur approximative de π) :

```
// une fonction sans argument
function getApproximationPi() {
    return 3.14; // ou 22/7
}
```

```
console.log(`360 deg = 2*pi ~= ${2 * getApproximationPi()} rad`);
```



```
360 deg = 2*pi ~= 6.28 rad
```

Une fonction qui appelle d'autres fonctions (ici retourne le résultat de l'appel à une autre fonction spécialisée, en fonction de l'opération demandée) :

```
// fonction qui retourne le résultat de l'addition de 2 nombres
function getResultatAdditionDe2Nombres(nombre1, nombre2) {
    return nombre1 + nombre2;
}

// fonction qui retourne le résultat de la multiplication de 2 nombres
function getResultatMultiplicationDe2Nombres(nombre1, nombre2) {
    return nombre1 * nombre2;
}
```

```
// une fonction qui appelle d'autres fonctions, en fonction de l'opération demandée
// et retourne le résultat reçu de la fonction appelée
// note : l'instruction "return" met fin à l'exécution de la fonction
function getResultatCalculEntre2Nombres(nombre1, operation, nombre2) {
    // en fonction de l'opération demandée
    switch (operation) {
        // addition et soustraction (a - b = a + -b)
        case "addition":
            return getResultatAdditionDe2Nombres(nombre1, nombre2);
        case "soustraction":
            return getResultatAdditionDe2Nombres(nombre1, -nombre2);

        // multiplication et division (a / b = a * 1/b)
        case "multiplication":
            return getResultatMultiplicationDe2Nombres(nombre1, nombre2);
        case "division":
            return getResultatMultiplicationDe2Nombres(nombre1, 1 / nombre2);

        // opérations non gérées par la fonction
        default:
            console.log(`Opération "${operation}" non gérée`);
            return null;
    }
}
```

```
console.log(`7 + 2 = ${getResultatCalculEntre2Nombres(7, "addition", 2)}`);  
console.log(`6 - 1 = ${getResultatCalculEntre2Nombres(6, "soustraction", 1)}`);  
console.log(`3 * 4 = ${getResultatCalculEntre2Nombres(3, "multiplication", 4)}`);  
console.log(`10 / 2 = ${getResultatCalculEntre2Nombres(10, "division", 2)}`);  
console.log(`2 ** 3 = ${getResultatCalculEntre2Nombres(2, "puissance", 3)}`);
```



7 + 2 = 9
6 - 1 = 5
3 * 4 = 12
10 / 2 = 5
Opération "puissance" non gérée
2 ** 3 = null

IX. Les tableaux

Les **tableaux** permettent de **stocker des données**, de n'importe quel type, de façon **groupée**.

Si des données à manipuler ont un **lien** entre elles, alors il est plus **pratique** d'utiliser un tableau et de l'affecter à une seule variable, plutôt que de les traiter séparément. De cette façon, **les données qui ont un lien entre elles restent groupées** et on peut facilement **automatiser** un traitement.

1. Les tableaux indexés

Leur **symbole** est le crochet « **[]** » (comme dans la plupart des langages).

Un tableau indexé **regroupe des éléments**.

C'est une liste **ordonnée d'éléments** (l'ordre des éléments est conservé).

Chaque élément a pour **emplacement** un « **index** » du tableau.

La longueur (« **length** » en anglais) d'un tableau indexé est le nombre d'éléments qu'il contient.

Attention, dans un tableau indexé contenant n éléments :

- le **premier** élément a l'**index 0 (zéro)**
- le **dernier** élément a l'**index (n - 1)**

a. Utilités

- permet de **regrouper** des éléments ensemble, dans le but par exemple de « **boucler sur le tableau** » (souvent grâce à une boucle « for ») et **appliquer une logique commune pour chaque élément**
- il est **extensible**, pouvant contenir **0 à n éléments**, donc très pratique quand on ne sait pas à l'avance sur combien d'éléments on devra appliquer une logique
- chaque élément peut être de **n'importe quel type** (on peut y stocker des nombres, des tableaux, des fonctions, etc.)

b. Les bases de leur utilisation

Voici les opérations les plus courantes sur les tableaux indexés.

Créer un tableau indexé vide (qui contient 0 élément) et l'affecter à une variable :

```
// initialiser un tableau vide  
const tableauVide = [];
```

Créer un tableau indexé qui contient des valeurs fournies (ici 7 nombres) et l'affecter à une variable :

```
const tabNums = [11, 22, 33, 44, 55, 66, 77];
```

Ajouter un élément à la fin du tableau (en dernière position) :

```
// ajouter un élément à la fin du tableau  
tableauVide.push(2);  
tableauVide.push(3);  
tableauVide.push(7);
```

```
console.log("tableauVide = ", tableauVide);
```



```
tableauVide =  ► (3) [2, 3, 7]
```

Récupérer un élément par son index (son emplacement, qui commence par 0) :

```
// récupérer un élément par son index  
console.log("tableauVide[1] = ", tableauVide[1]);
```



```
tableauVide[1] = 3
```

Modifier la valeur d'un élément à un certain index :

```
// modifier la valeur de l'élément à l'index 2  
tableauVide[2] = "abc";
```

```
console.log("tableauVide = ", tableauVide);
```



```
tableauVide = ▶ (3) [2, 3, 'abc']
```

Boucler sur un tableau et appliquer une logique pour chaque élément :

```
// boucler sur un tableau (de 0 à length-1)  
for (let i = 0; i < tableauVide.length; i++) {  
  // récupération de l'élément à l'index i  
  const elementTableauVide = tableauVide[i];  
  // log de cet élément  
  console.log("tableauVide[" + i + "] = ", elementTableauVide);  
}
```



tableauVide[0]	=	2
tableauVide[1]	=	3
tableauVide[2]	=	abc

c. Quelques méthodes utiles

Une **méthode** est une **fonction qui appartient à une classe**. Comme les fonctions en général, une méthode peut avoir besoin d'argument(s) et peut retourner un résultat.

Les tableaux sont de la **classe Array**.

Syntaxe : pour appeler une méthode « uneMethode » sur un tableau indexé « unTableau », on écrit :

```
unTableau.uneMethode()
```

Voici quelques **méthodes utiles sur les tableaux** indexés (classe Array) :

.push(newElement) : **ajouter un élément à la fin du tableau** (déjà vu plus haut dans les opérations courantes)

```
// ajouter un élément à la fin du tableau  
tableauVide.push(2);
```

.join(separateur) : transforme un **tableau en string**, en **concaténant** tous les éléments dans l'ordre et en ajoutant le **séparateur entre chaque élément**

```
// tabNums transformé en string avec "_|" entre chaque élément  
console.log(`tabNums.join("_|") = ${tabNums.join("_|")}`);
```



```
tabNums.join("_|") = 11_|_22_|_33_|_44_|_55_|_66_|_77
```

.includes(elementRecherche) : renvoie true ou false, si l'**élément** est **présent ou non dans le tableau** (ici on cherche l'élément 77 puis 42 dans le tableau tabNums) :

```
// .includes(elementRecherche) : renvoie true/false si l'élément
est présent ou non dans le tableau
console.log(
  `L'élément ${77} ${tabNums.includes(77) ? "est" : "n'est pas"}
  présent dans le tableau [${tabNums.join(", ")}]`
);
console.log(
  `L'élément ${42} ${tabNums.includes(42) ? "est" : "n'est pas"}
  présent dans le tableau [${tabNums.join(", ")}]`
);
```



```
L'élément 77 est présent dans le tableau [11, 22, 33, 44, 55, 66, 77]
L'élément 42 n'est pas présent dans le tableau [11, 22, 33, 44, 55, 66, 77]
```

.indexOf(elementRecherche) : renvoie l'**index de l'élément recherché** (entre 0 et length-1 si trouvé, -1 sinon) :

```
// .indexOf(elementRecherche) : renvoie l'index de l'élément
recherché (entre 0 et length-1 si trouvé, -1 sinon)
const indexOf77 = tabNums.indexOf(77);
if (indexOf77 !== -1) {
  console.log(`L'élément ${77} est à l'index ${indexOf77} dans
  le tableau [${tabNums.join(", ")}]`);
} else {
  console.log(`L'élément ${77} n'est pas présent dans le tableau
  [${tabNums.join(", ")}]`);
}
```



```
L'élément 77 est à l'index 6 dans le tableau [11, 22, 33, 44, 55, 66, 77]
```

.slice(indexDebutEltInclus, indexFinEltExclu) : renvoie une **copie** (sans impacter le tableau initial) **d'un morceau du tableau** (attention l'index de début est inclus mais l'index de fin est exclu) :

```
const morceau = tabNums.slice(2, 4);
console.log(`morceau (tabNums.slice(2, 4)) = [${morceau.join(", ")}]`);
```



```
morceau (tabNums.slice(2, 4)) = [33, 44]
```

2. Les tableaux associatifs

Leur **symbole** est l'**accolade** « {} » (peut varier en fonction des langages).

Attention donc à ne pas confondre un tableau associatif avec le corps d'un bloc, d'une fonction ou autre, puisqu'ils ont le même symbole. Bien **indenter** votre code vous aidera !

Un **tableau associatif** regroupe des éléments sous la forme de **paires** (ou duos) **clef/valeur**.

Un tableau associatif contient 0 à n clefs et chaque clef pointe vers une valeur (une donnée de n'importe quel type).

Une façon simple de se représenter un **tableau associatif** est de le **comparer à un dictionnaire**. Les **clefs** sont les **mots** et chaque mot pointe vers une **valeur** qui est la **définition** de ce mot.

a. Utilités

- permet de **regrouper des données**, de façon **structurée**, dans le but de « **construire un objet** » qu'on pourra **facilement manipuler**
- il est **extensible**, pouvant contenir 0 à n paires clef/valeur
- chaque **valeur** peut être de **n'importe quel type** (on peut y stocker des nombres, des tableaux, des fonctions, etc.)

b. Les bases de leur utilisation

Voici les opérations les plus courantes sur les tableaux associatifs.

Création d'un tableau associatif vide (sans aucune paire clef/valeur) :

```
// initialisation d'un tableau associatif vide
let tabAssocVide = {};
```

Ajout d'une paire clef/valeur (ici le prénom Bryan et l'âge 27) :

```
tabAssocVide.prenom = "Bryan";
tabAssocVide.age = 27;
```

```
console.log("tabAssocVide = ", tabAssocVide);
```



```
tabAssocVide = {prenom: 'Bryan', age: 27}
```

Accéder à une valeur (ici la propriété « prenom ») :

```
console.log(`Je m'appelle ${tabAssocVide.prenom}`);
```



```
Je m'appelle Bryan
```

Modification d'une valeur (ici l'âge passe à 28) :

```
tabAssocVide.age = 28;  
console.log("tabAssocVide = ", tabAssocVide);
```



```
tabAssocVide = ▶ {prenom: 'Bryan', age: 28}
```

Créer un tableau associatif contenant des paires clef/valeur fournies (ici le Pokémon Pikachu) :

```
// initialisation d'un tableau associatif plus riche  
// affectation à la variable pikachu, stockant ses infos et ses attaques  
// (données sur Pikachu récupérées d'internet)  
let pikachu = {  
  id: 25,           // number (entier)           : identifiant unique  
  name: "Pikachu", // string                    : nom du Pokemon  
  weightKg: 6.0,    // number (décimal)          : poids  
  hpMax: 80,        // number (entier)            : points de vie max  
  attacks: [        // Array (tableau indexé) : liste des attaques  
    {  
      name: "Vive-Attaque", // string          : nom de l'attaque  
      damages: 10,         // number (entier) : puissance de l'attaque  
    },  
    {  
      name: "Boule Élek",  
      damages: 20,  
    },  
  ],  
};
```

```
console.log("pikachu = ", pikachu);
```



```
pikachu = ▶ {id: 25, name: 'Pikachu', weightKg: 6, hpMax: 80, attacks: Array(2)}
```



```
pikachu = {id: 25, name: 'Pikachu', weightKg: 6, hpMax: 80, attacks: Array(2)}
  attacks: Array(2)
    0: {name: 'Vive-Attaque', damages: 10}
    1: {name: 'Boule Élek', damages: 20}
    length: 2
    [[Prototype]]: Array(0)
  hpMax: 80
  id: 25
  name: "Pikachu"
  weightKg: 6
  [[Prototype]]: Object
```

3. Exemples d'application

a. Tableaux indexés et associatifs avec Pikachu

En se basant sur la **structure du Pokémon Pikachu**, écrire dans la console :

```
Pikachu possède 2 attaques :  
    "Vive-Attaque" (puissance 10)  
    "Boule Élek" (puissance 20)
```

Le **code** devra pouvoir **s'adapter** à d'autres Pokémon (tableaux associatifs) qui suivront la **même structure**.

Note : pour faire une tabulation / un retrait dans la console, écrire le caractère spécial `"\t"`.

```
// création d'une fonction, ainsi il sera simple d'appeler cette logique avec d'autres Pokemons  
function listerAttaquesPokemon(pokemon) {  
    // écriture de la première ligne  
    console.log(`${pokemon.name} possède ${pokemon.attacks.length} attaques :`);  
  
    // pour chaque attaque (dans le tableau indexé pokemon.attacks)  
    for (let i = 0; i < pokemon.attacks.length; i++) {  
        // récupération du tableau associatif de l'attaque (le i-ème élément)  
        const attaque = pokemon.attacks[i];  
  
        // écriture d'une ligne d'attaque  
        console.log(`\t"${attaque.name}" (puissance ${attaque.damages})`);  
    }  
}  
  
// appel de la fonction listerAttaquesPokemon en lui passant pikachu comme argument/paramètre  
listerAttaquesPokemon(pikachu);
```



```
Pikachu possède 2 attaques :  
    "Vive-Attaque" (puissance 10)  
    "Boule Élek" (puissance 20)
```


X. DOM

1. Définition

Selon Wikipédia :

Le Document Object Model (DOM) est une interface de programmation normalisée par le W3C, qui permet à des scripts d'examiner et de modifier le contenu du navigateur web. Par le DOM, la composition d'un document HTML ou XML est représentée sous forme d'un jeu d'objets – lesquels peuvent représenter une fenêtre, une phrase ou un style, par exemple – reliés selon une structure en arbre. À l'aide du DOM, un script peut modifier le document présent dans le navigateur en ajoutant ou en supprimant des nœuds de l'arbre.

Autrement dit, le DOM est l'interface par laquelle JavaScript passera pour atteindre, ajouter, modifier ou supprimer les balises d'une page HTML. Chaque balise est un objet `HTMLElement` contenant les propriétés de style, de position, de taille et de comportement de la balise.

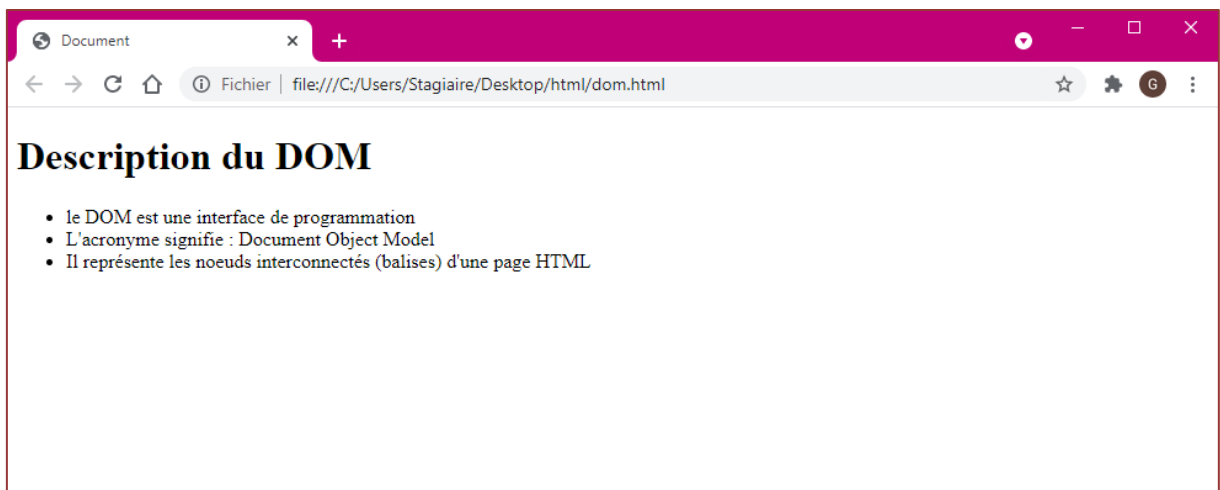
2. En pratique

Ainsi, l'exemple suivant représente un titre et une liste à puces au sein d'une page HTML 5 valide, vue sous différents aspects :

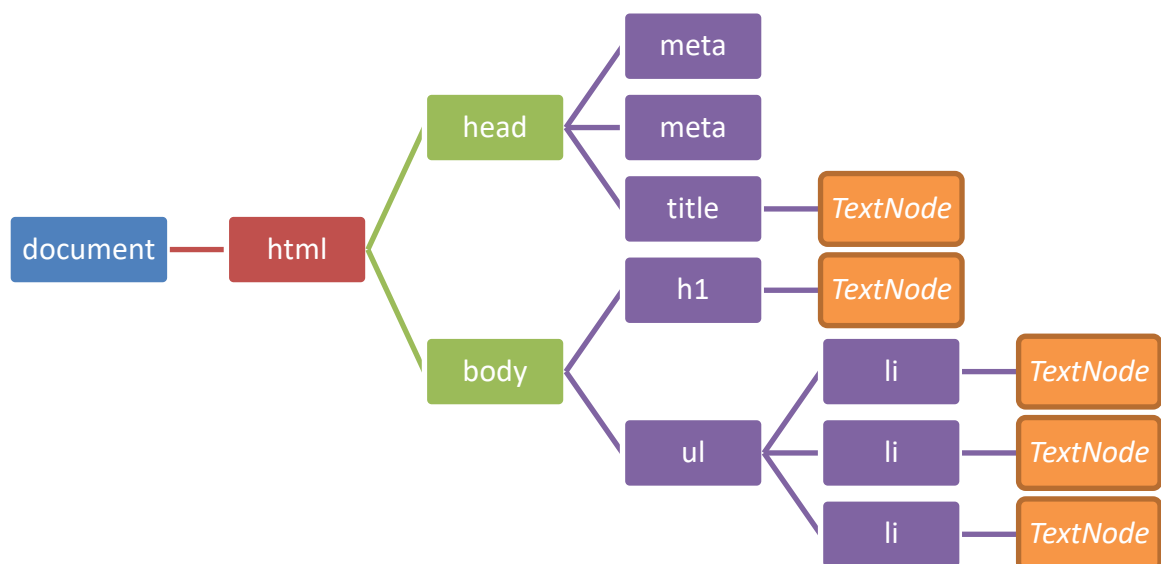
Le code du fichier :

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <title>Document</title>
7  </head>
8  <body>
9      <h1>Description du DOM</h1>
10     <ul>
11         <li>
12             le DOM est une interface de programmation
13         </li>
14         <li>
15             L'acronyme signifie : Document Object Model
16         </li>
17         <li>
18             Il représente les noeuds interconnectés (balises) d'une page HTML
19         </li>
20     </ul>
21 </body>
22 </html>
```

L'interprétation du navigateur (page affichée) :



Et sa représentation sous forme d'arbre (DOM Tree) :



Ainsi, pour atteindre un élément spécifique, JavaScript doit démarrer par le "document" (élément racine de la page) et parcourir l'arbre jusqu'à atteindre la cible. Une fois cette cible atteinte et stockée dans une variable, JavaScript peut depuis ce point se déplacer vers le parent ou les enfants de celui-ci.

a. Atteindre des éléments du DOM

Par exemple, si nous voulons atteindre la liste (ul) de la page, nous écrirons :

```

22     <script>
23
24         let list = document.querySelector("ul")
25
26     </script>

```

La méthode **querySelector()** de l'objet document permet de pointer le premier élément répondant à la requête spécifiée dans l'argument sous forme de chaîne de caractères. Cette requête s'écrit de la même manière qu'une règle CSS.

Néanmoins, si plusieurs ul étaient présentes sur la page, seule la première dans l'arbre sera pointée.

De ce fait, si nous souhaitons récupérer tous les éléments de la liste, voici le code :

```

22     <script>
23
24         let list = document.querySelector("ul")
25
26         let listElements = list.querySelectorAll("li")
27
28     </script>

```

Ici nous utilisons la méthode **querySelectorAll()** en partant de la variable list (et non plus du document) afin de **récupérer tous les éléments enfants de CETTE ul** répondant à la requête. Jetons un œil sur ce que la console nous présente si nous vérifions les valeurs de ces deux variables :

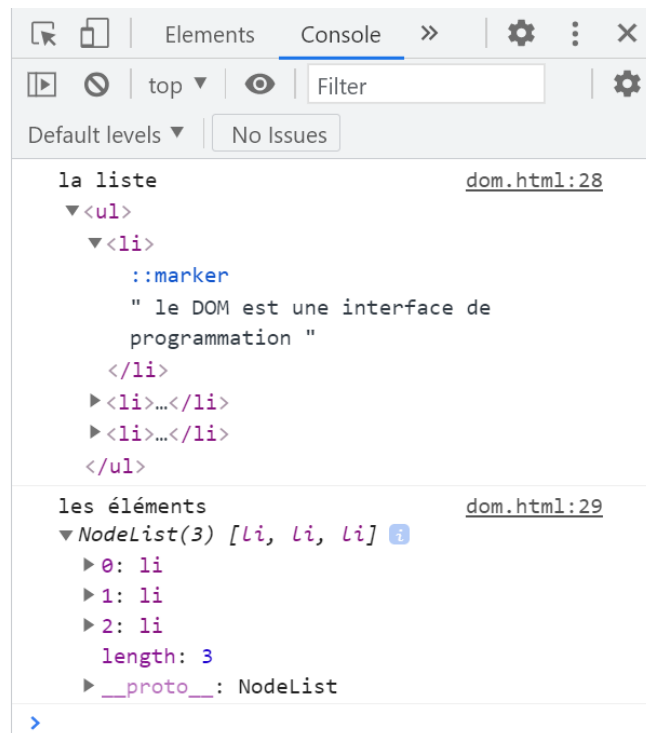
```

22     <script>
23
24         let list = document.querySelector("ul")
25
26         let listElements = list.querySelectorAll("li")
27
28         console.log("la liste", list)
29         console.log("les éléments", listElements)
30
31     </script>

```

Description du DOM

- le DOM est une interface de programmation
- L'acronyme signifie : Document Object Model
- Il représente les noeuds interconnectés (balises) d'une page HTML



La variable `list` présente une balise `ul` accompagnée de tous les nœuds enfants qu'elle contient. On constate également que chaque `li` à l'intérieur est composée d'un pseudo-élément `::marker` (la puce) et d'un **nœud de texte** (entre guillemets).

La variable `listElements` est sensiblement différente : elle comporte un objet `NodeList` contenant trois entrées (les trois `li`) dans un tableau indexé.

b. Modifier des éléments du DOM

Nous pourrions effectuer une boucle sur la `NodeList` "listElements" pour, par exemple, les modifier une par une. Faisons cela pour changer la couleur du texte de chaque `li` en rouge :

```
22 <script>
23
24   let list = document.querySelector("ul")
25
26   let listElements = list.querySelectorAll("li")
27
28   listElements.forEach(function(element){
29     element.style.color = "red"
30   })
31
32 </script>
```

Description du DOM

- le DOM est une interface de programmation
- L'acronyme signifie : Document Object Model
- Il représente les noeuds interconnectés (balises) d'une page HTML

JavaScript nous a ici permis de modifier la propriété `style` de chaque élément en effectuant une boucle `forEach()` sur la `NodeList`.

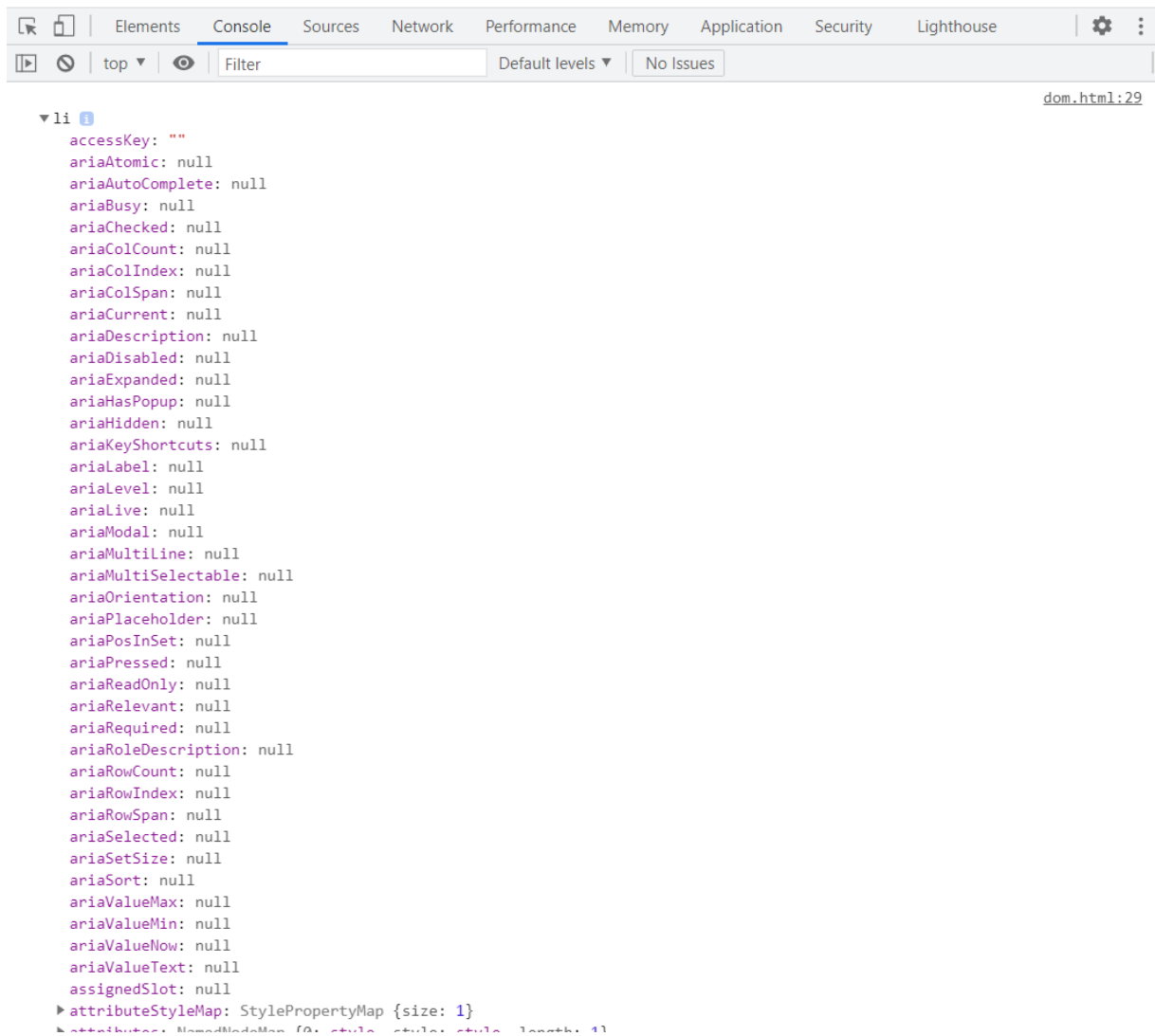
Pour rappel :

forEach prend en paramètre une fonction dite "callback" (ou fonction de rappel) qui sera exécuté à chaque tour de boucle. Dans cette fonction anonyme, chaque li sera représenté par la variable passée en argument "element" (qui peut être nommée autrement si besoin).

Comment connaître les propriétés d'un élément du DOM (comme ici, style) et ce qu'elles contiennent ? La commande **console.dir()**¹ vous permet de les consulter :

```
22     <script>
23
24         let list = document.querySelector("ul")
25
26         let listElements = list.querySelectorAll("li")
27
28         listElements.forEach(function(element){
29             console.dir(element)
30             element.style.color = "red"
31         })
32
33
34     </script>
```

¹ Là où `console.log()` présenterait l'élément de manière interactive, à la manière de l'inspecteur d'éléments du navigateur, `console.dir()` listera ses composantes d'objet JavaScript.



L'objet `HTMLElement` représentant cette balise `li` se compose de très nombreuses propriétés, comme évoqué plus haut, définissant toutes les valeurs nécessitées par le navigateur pour l'interpréter correctement (son style, sa taille, sa position sur la page, son contenu...). En descendant un peu dans cette longue liste, nous trouvons par exemple :

```
outerHTML: "<li style=\"color: red;\">\n          le DOM est une interface de programmation\n        </li>"
outerText: "le DOM est une interface de programmation"
ownerDocument: document
parentElement: ul
parentNode: ul
part: DOMTokenList [value: ""]
prefix: null
previousElementSibling: null
previousSibling: text
scrollHeight: 18
scrollLeft: 0
scrollTop: 0
scrollWidth: 297
shadowRoot: null
slot: ""
spellcheck: true
style: CSSStyleDeclaration {0: "color", additiveSymbols: "", alignContent: "", alignItems: "", alignSelf: "", alignment...
tabIndex: -1
tagName: "LI"
textContent: "\n          le DOM est une interface de programmation\n        "
title: ""
```

- `outerHTML` et `outerText` : ce sont les textes représentant le li soit en HTML, soit uniquement sa représentation textuelle, tel que le navigateur l'a reçu pour interprétation.
- `parentNode` : l'élément du DOM parent de ce li (nous voyons donc "ul")
- `style` : un objet `CSSStyleDeclaration` contenant l'intégralité des propriétés CSS du li et leurs valeurs. C'est cela qui a été atteint dans le code JavaScript précédent, pour modifier dans cet objet la propriété `color` et lui assigner la valeur "red".
- `tagName`: le nom de la balise utilisée pour cet élément (li)

Naturellement, il y a beaucoup trop de propriétés dans un `HTMLElement` pour les lister toutes ici, mais il faut savoir et acquérir le réflexe de se renseigner en utilisant la console, afin de déterminer si notre code JavaScript effectue les bonnes actions sur les bonnes propriétés.