

CHAPITRE 1 : STRUCTURE LEXICALE

JavaScript est sensible à la case. Sa syntaxe est composée des commentaires, des littéraux, des identifiants, des mots réservés ainsi que des caractères Unicode.

Les commentaires s'insèrent en rajoutant du texte sur la même ligne après une séquence « `//` » ou sur une ou plusieurs lignes entre « `/*` » et « `*/` ».

Un *littéral* est une séquence d'un ou de plusieurs caractères utilisés pour représenter la valeur d'une donnée intrinsèque dans un script JavaScript. Il peut représenter un nombre, une donnée booléenne (`true` ou `false`), un texte, une indécision (`NaN`) ou l'absence de valeurs (`null` ou `undefined`).

Un *identifiant* est un nom. Il permet d'utiliser des constantes, des variables, des propriétés, des fonctions et classes. Il permet aussi d'étiqueter certaines séquences de code. Un identifiant doit commencer avec un caractère alphabétique (ex : `a`), un caractère de soulignement « `_` », ou un symbole dollar « `$` ». Peuvent s'en suivre des caractères alphabétiques, des nombres, des caractères de soulignement ou des caractères dollar. Un identifiant ne doit pas contenir d'emojis ou d'autres symboles dans sa syntaxe. Par exemple, `$hello`, `_8bits` et `maman` sont des identifiants valides mais pas `Euro` ou `20Personnes`.

Les *mots réservés* sont des mots qui font partie du lexique de JavaScript. On peut citer `as`, `const`, `let` comme exemples. Ils ne doivent pas être utilisés pour un contexte autre que celui qui leur est attribué par le langage. Pour la liste complète des mots réservés, voir « *JavaScript : The definitive guide* ».

Les scripts JavaScript sont encodés avec le jeu de caractères de l'*Unicode*. On peut utiliser n'importe quel caractère de l'Unicode pour écrire du JavaScript valide. On peut les écrire de manière directe (ex : `élève`) ou à l'aide des séquences d'échappement – on place le code Unicode correspondant au caractère souhaité dans « `\u{ici}` » ou « `\U{ici}` » (ex : `\u{00e9}l\u{00e8}ve`).

CHAPITRE 2 : TYPES, VALEURS ET VARIABLES

Le *type* est la nature d'une valeur. En JavaScript, il y a deux catégories de types : les **types primitifs** et les **objets**.

Un type primitif décrit la nature d'une valeur intrinsèque et immuable. Les nombres, le texte, les valeurs booléennes, `null`, `undefined` et `NaN` sont des valeurs de type primitif. Toute valeur non-intrinsèque est un objet (ex : un tableau, un set, une map, une date). Un objet est une collection de *propriétés* où chaque propriété est composée d'un *nom* et d'une *valeur* associée qui peut elle aussi être de nature primitive ou s'agir d'un autre objet. Un objet est mutable, c'est un peu comme une *structure* en C, mais amélioré.

Javascript suit un style orienté objet. C'est-à-dire que chaque valeur a, de par son type, des fonctions qui opèrent avec elle (ex : `'Hello'.trim()`). Exception est uniquement faite pour `null` et `undefined`.

TYPES PRIMITIFS : LES NOMBRES

Un *nombre* (`number`), en JavaScript est représenté par un ou plusieurs caractères numériques et permet de dénoter un élément de l'ensemble \mathbb{N} (entiers naturels) ou \mathbb{R} (nombres à virgules flottantes).

Un entier naturel en JavaScript s'écrit de quatre formes :

1. De par sa forme décimale (ex : `10`, `3`, `01`) ;
2. De par sa forme hexadécimale : « `0xhexa` » ou « `0Xhexa` » (ex : `0x2`, `0xf`) ;
3. De par sa forme octale : « `0oocta` » ou « `00octa` » (ex : `0o13`, `007`) ;
4. De par sa forme binaire : « `0bbin` » ou « `0Bbin` » (ex : `0b1101`, `0B101`).

On peut écrire n'importe quel nombre entre `-9007199254740992` et `9007199254740992`.

Un *nombre à virgule flottante* en JavaScript s'écrit de deux formes :

1. Avec une virgule flottante : « `nombres.nombres` » ou « `.nombres` » (ex : `3.2`, `.05`). Lorsqu'un nombre à virgule flottante commence par un point, JavaScript suppose qu'il y a un zéro avant la virgule (ex : `.2` signifie `0.2`) ;
2. Sous la forme d'une puissance de 10 : « `a.be±c` », « `a.bE±c` » (ex : `7.3e-9`, `0.5E10`, `6.7E+2`) ou « `.be±c` », « `.bE±c` » (ex : `.1e-2`, `.1E6`, `.7E+2`). Le « `E±c` » veut simplement dire « $\times 10^{\pm c}$ ».

Les nombres à virgule flottante en JavaScript ne s'écrivent qu'en puissance de 10. On peut écrire n'importe quel nombre entre $\pm 5 \times 10^{-324}$ et $\pm 1.7976931348623157 \times 10^{308}$.

On peut utiliser des « `_` » entre les chiffres composant un nombre entier naturel ou à virgule flottante pour en faciliter la lisibilité (ex : `1_000_000_000` équivaut à `1000000000`).

D'autres littéraux permettent de dénoter certaines données de nature numérique :

1. Le littéral `Infinity` permet de représenter l'infini. Il est souvent le résultat de certaines opérations comme la division d'un nombre non-nul par zéro, ou l'évaluation d'un nombre supérieur (*overflow*) ou inférieur (*underflow*) à la limite assignable (ex : `5E308*2`).
2. Le littéral `NaN` permet de représenter une valeur numérique indéfinie. Il est le résultat de la division de zéro par zéro ou de l'infini par l'infini. Deux ou plusieurs `NaN` sont mutuellement exclusifs.

TYPES PRIMITIFS : LE TEXTE

Un texte (string) en JavaScript est représenté par une séquence de caractères alphanumériques et de symboles délimités par deux crochets (`'`), deux double-crochets (`"`) ou deux accents graves (```).

Lorsqu'on utilise deux crochets pour délimiter du texte, on peut y placer autant de double-crochets et d'accents graves que l'on veut, ils seront considérés comme du texte simple (ex : `'écrire ` et " OK'`).

Lorsqu'on utilise deux double-crochets pour délimiter du texte, on peut y placer autant de crochets ou d'accents graves que l'on veut, ils seront considérés comme du texte simple (ex : `"écrire ' et ` OK"`).

Lorsqu'on utilise deux accents graves pour délimiter du texte, on peut y placer autant de crochets ou de double-crochets que l'on veut, ils seront considérés comme du texte simple (ex : ``I said: "OK``).

Ecrire du texte avec des accents graves permet aussi d'évaluer des expressions JavaScript à l'intérieur du corps de ce texte (ex : ``j'ai ${300+50} francs`` sera évaluée à ``j'ai 350 francs``). Tout ce qui se trouve dans « `$(ici)` » sera évaluée comme une expression et son résultat sera incorporée au texte lors de l'interprétation du script.

On peut aussi placer des séquences d'échappement dans du texte. Leur structure est : « `\échappement` » (ex : `'J\\'ai dit \"OK\"'`). Pour une liste complète des caractères d'échappements, voir « *JavaScript : The definitive guide* ».

Utiliser des séquences d'échappement est très utile quand on veut placer des apostrophes, des guillemets ou des accents graves dans son texte mais que les délimiteurs utilisés nous en empêchent. Elles nous permettent aussi d'insérer certains caractères de formatage (ex : les retours à la ligne) qui sont normalement ignorées par l'interpréteur.

On peut aussi utiliser deux slashes (`/`) pour délimiter du texte. Dans ce cas, le texte qui s'y trouve est une expression régulière (ex : `/^Je/`).

TYPES PRIMITIFS : LES VALEURS BOOLEENNES

Une valeur booléenne (boolean) est représentée par deux littéraux : « `true` » et « `false` ». Ces littéraux expriment respectivement la vérité et la fausseté.

Les valeurs booléennes en JavaScript sont souvent les résultats d'opérations de comparaison. De ce fait, elles sont plus utilisées dans les structures de contrôle (`if`, `for`, etc.).

TYPES PRIMITIFS : NULL ET UNDEFINED

L'absence de valeurs ou le vide s'exprime grâce à deux littéraux : « `null` » et « `undefined` ».

Le littéral « `undefined` » est la valeur stockée par défaut dans une variable, une constante ou un objet lorsqu'elle est déclarée mais non-initialisée. Elle est aussi le résultat de l'appel d'une propriété non-définie d'un objet. Le littéral « `null` », lui, n'est généralement pas résultat d'une quelconque opération. Ce littéral est souvent utilisé par les développeurs pour exprimer de manière explicite qu'une variable ne contient aucune valeur.

De par ce qu'ils représentent, il n'y a aucune différence entre les deux, mais de par leurs types respectifs, ils sont différents. `undefined` est la propriété d'un objet qui porte le même nom et qui a une valeur indéfinie, alors que `null` est un mot-clé spécifiant une constante définie par *l'objet global*.

L'OBJET GLOBAL

Avant qu'un script ne soit interprété, JavaScript définit une série de constantes, de fonctions et d'objets qu'il rend directement accessibles dans le script. Il stocke toutes ces valeurs dans un objet qu'on appelle *l'objet global*, dénoté : `globalThis`. Cet objet nous permet d'utiliser des fonctions natives du langage ainsi que certaines API de son environnement (comme l'API **DOM** du côté client qui définit par exemple la fonction `console.log()` ou l'API **module** dans `Node.js` qui définit la fonction `require()`).

Pour utiliser une propriété de l'objet global, il suffit de l'appeler là où on veut dans le script (ex : `Infinity`).

Donc, à chaque fois que vous rencontrez dans un script une constante, un objet ou une fonction qui n'a pas préalablement été définie dans le script même ou rajoutée comme module, dites-vous qu'il s'agit d'une propriété de l'objet global.

L'ensemble de ces propriétés constituent la *bibliothèque standard* du JavaScript. Il est à noter que cette bibliothèque est différente en fonction de l'environnement (côté client, elle contient aussi les différentes API du navigateur. Côté serveur, dans `Node.js` par exemple, elle contient aussi les différentes API du serveur).

LES VARIABLES

Une variable est utilisée dans un script lorsqu'on a besoin de stocker une valeur en mémoire en vue de l'utiliser ultérieurement dans le même script ou dans un autre si on l'exporte dans un *module*. La création d'une variable se déroule en deux étapes (qui peuvent être effectuées de manière simultanée ou différée) : la *déclaration* et l'*initialisation*.

On peut déclarer une variable de trois manières :

1. Avec le mot-clé `let` : Cela signifie que la variable peut contenir une valeur à un moment, puis une autre, au gré de ce que le développeur veut en faire (ex : `let identifiant;`).
2. Avec le mot-clé `var` : Similaire à la première manière (ex : `var identifiant;`). La seule différence se trouve dans sa portée. Une variable déclarée avec `var` peut être utilisée avant d'être déclarée.
3. Avec le mot-clé `const` : Dans ce cas, on l'appelle « *Constante* ». Cela signifie qu'elle ne peut contenir que la valeur qui lui est attribuée lors de son initialisation (ex : `const pi = 3.14;`). De ce fait, une constante doit toujours être déclarée et initialisée au même moment.

Lorsqu'on déclare une variable sans l'initialiser, JavaScript lui affecte la valeur « `undefined` » par défaut. On peut l'initialiser en même temps que sa déclaration ou bien ultérieurement dans sa zone de portée (ex : `const pi = 3.14; //déclaration et initialisation simultanées`
`let g; //déclaration d'une autre variable g`
`g = 9.8; //initialisation de g après sa déclaration`).

On peut aussi créer plusieurs variables en une seule fois. Il suffit d'écrire le mot-clé suivi des identifiants des variables que l'on veut créer séparés par des virgules (ex : `let a1 = '', a2, a3 = 2;`). Enfin, on peut

aussi utiliser la notation de **déstructuration** pour déclarer et initialiser plusieurs variables en une seule expression (ex :

```
let [a,b] = [2,1]; //cela déclare et initialise a à 2 et b à 1
let {a,b} = {a:2, b:1}; //idem, mais en utilisant un objet).
```

On utilise les variables comme opérandes dans les expressions. Pour utiliser une variable préalablement créée, il suffit d'écrire son nom dans une expression se trouvant dans sa portée.

La portée d'une variable est une zone dans un script JavaScript dans laquelle on peut utiliser une variable préalablement déclarée. Elle dépend du mot-clé utilisé pour créer la variable. Si on a déclaré la variable avec `let` ou `const` et que cette déclaration se trouve en dehors de tout bloc ou fonction, La portée de cette variable est dite *globale*. En effet, cette variable sera accessible n'importe où dans le script (ex :

```
a = 3; //Erreur! Quand on utilise let, on ne peut pas utiliser a avant sa déclaration
let a = 71; //déclaration d'une variable globale
a; //elle est accessible ici
{
  a; //ici aussi
}
a; //ici aussi. Bref, partout).
```

Si, par contre, la déclaration se trouve dans un bloc ou dans une fonction, elle est dite locale à cette fonction ou à ce bloc. On ne peut utiliser cette variable que dans cette fonction ou dans ce bloc (ex :

```
{
  let a = 71; //déclaration d'une variable de portée locale dans un bloc
  a; //on peut accéder à la variable a ici
  {
    a; //ici, a est globale par rapport à cet autre bloc. Accessible
  }
}
a; //mais pas ici, car elle est en dehors du bloc, donc hors de portée).
```

Cela va de même si on déclare la variable avec le mot-clé `var`. Cependant, utiliser cette dernière à une propriété appelée *Hoisting*. Cette propriété permet d'utiliser une variable avant ou même sans la déclarer, en effet lorsque le script est interprété (JavaScript s'interprète de haut en bas), toutes les déclarations utilisant le mot-clé `var` ainsi que toutes les variables utilisées mais jamais déclarées sont redéclarées vers le haut du bloc ou de la fonction dans laquelle elles se trouvent (ex : le code suivant :

```
maVar1 = 4; //utilisation d'une variable non-déclarée
{
  maVar2 = 5; //utilisation d'une variable non-déclarée
}
var maVar1 = 1; //déclaration de la variable préalablement utilisée
                //Remarquez que maVar2 est utilisée mais jamais déclarée
```

ressemblera à ceci après que l'interpréteur ait effectué le hoisting :

```
let maVar1; //maVar1 est redéclarée en première ligne du script
maVar1 = 4;
{
  let maVar2; //maVar2 est redéclarée en première ligne du bloc où elle est utilisée
  maVar2 = 5;
}
maVar1 = 1;). Cependant, il est recommandé de toujours utiliser let pour déclarer une variable.
```

On ne peut pas déclarer une même variable plusieurs fois dans une même portée (ex :

```
let a = 2; //déclaration d'une variable
let a = 7; //Erreur! On ne peut pas déclarer a car elle l'est déjà dans cette portée
{
  let a = 5; //On peut redéclarer a dans un bloc car sa portée sera locale
  a; //ceci appelle a=5
}
a; //ceci appelle a=2, car a=5 est hors de portée ici).
```

CHAPITRE 3 : EXPRESSIONS ET OPERATEURS

Une expression est tout ce dont l'évaluation, après interprétation, produit comme résultat, une valeur. Elle est composée d'*opérandes* et d'*opérateurs*. Il existe 12 types d'expressions :

EXPRESSIONS : EXPRESSIONS PRIMAIRES

Une expression primaire est une expression qui ne consiste qu'en un seul opérande. Cet opérande peut s'agir d'un littéral (ex : 4), d'un nom de variable, de **null**, de **this**, ou d'une propriété de l'objet global. Une expression primaire seule n'est pas évaluée. Elle est plutôt utilisée comme un opérande dans d'autres expressions.

EXPRESSIONS : EXPRESSIONS D'INITIALISATION DE TABLEAUX ET D'OBJETS

Une expression d'initialisation de tableau consiste en une liste d'expressions séparées par des virgules et délimitées par deux crochets. Sa syntaxe est donc : [*expr*, *expr*, *expr*, *ainsi de suite*], où *expr* s'agit de n'importe lequel des 12 types d'expressions.

L'évaluation d'une expression d'initialisation de tableau a comme résultat, la création d'un tableau dont les éléments sont les résultats des expressions listés. Ce tableau peut ensuite être stocké en mémoire en étant assignée à une variable. On utilise cette expression pour créer des tableaux ou pour déclarer plusieurs variables en une seule ligne via la notation de déstructuration (ex :

```
/*on écrit une expression d'initialisation de tableau avec quatre éléments : un nombre, un texte, une constante de l'objet global et un autre tableau et on l'affecte à la variable monTableau, créant ainsi un tableau*/
let monTableau = [11,"1", undefined, [null, this]];
/*on déclare et initialise 4 variables à l'aide de la notation de déstructuration. a contient un nombre ; b, du texte ; c, undefined et d, un tableau de deux éléments*/
let [a,b,c,d] = [11,"1", undefined, [null, this]];
```

Une expression d'initialisation d'objet consiste en une liste de propriétés séparées par des virgules et délimitées par deux accolades, sachant qu'une propriété est constituée d'un nom et d'une autre expression. Sa syntaxe est : {*nom* : *expr*, *nom* : *expr*, *nom* : *expr*, *ainsi de suite*}, où chaque *nom* s'agit d'un identifiant et chaque *expr* s'agit de n'importe lequel des 12 types d'expressions.

L'évaluation d'une expression d'initialisation d'objet a comme résultat, la création d'un objet dont les propriétés ont comme valeurs les résultats des expressions listés. Cet objet peut ensuite être stocké en mémoire en étant assignée à une variable. On utilise cette expression pour créer des objets ou pour déclarer plusieurs variables en une seule ligne via la notation de déstructuration (ex :

```
/*on écrit une expression d'initialisation d'objet avec cinq éléments et on l'affecte à la variable monObjet, créant ainsi un objet*/
let monObjet = {
  unNombre : 11,
  unTexte : `1`,
  indefini : undefined,
  unTableau : [2,3],
  unAutreObjet : {bla : "j'suis là"}
};
/*on crée 5 variables à l'aide de la notation de déstructuration : a contient un nombre ; b, du texte ; c, undefined ; d, un tableau de deux éléments ; e, un objet*/
let {a,b,c,d,e} = {
  unNombre : 11,
  unTexte : `1`,
  indefini : undefined,
  unTableau : [2,3],
  unAutreObjet : {bla : "j'suis là"}
};
```

EXPRESSIONS : EXPRESSIONS DE DEFINITION DE FONCTION

Une expression de définition de fonction permet, comme son nom l'indique, de définir une *fonction*. Le résultat de cette expression est une fonction. Ça peut paraître bizarre de dire ça, mais en JavaScript, une fonction est une valeur. Donc, il a un type : l'objet `Function`. C'est l'équivalent d'un pointeur sur fonction en C. Cela nous permet de la passer comme argument à d'autres fonctions, de la retourner comme valeur de retour d'autres fonctions ou de la stocker dans une variable). Il y a 2 syntaxes :

1. `function (paramètres) bloc` ou
2. `(paramètres) => instructions`

Toutes ces syntaxes seront expliquées au chapitre 7, donc pas d'exemples ici. Les expressions de définition de fonction permettent de faire de la *programmation déclarative*.

EXPRESSIONS : EXPRESSIONS D'ACCES A LA PROPRIETE D'UN OBJET

Un objet est une collection de propriétés. Et chaque propriété est constituée d'un nom et d'une valeur. On utilise une expression d'accès à la propriété d'un objet pour justement accéder à la valeur liée à cette propriété. Le résultat de cette expression est la valeur qui correspond à cette propriété. Il y a 2 syntaxes :

1. `nomObjet.nomPropriété` ou
2. `nomObjet["nomPropriété"]`

En JavaScript, un tableau est aussi un objet. La seule différence est que pour chaque propriété, le nom est remplacé par un *index* (un numéro unique attribué automatiquement) et d'une valeur. L'expression d'accès à la valeur d'un élément d'un tableau est donc : `nomTableau[index]`.

On en parlera en profondeur dans les chapitres 5 et 6.

EXPRESSIONS : APPELS OU INVOCATIONS

Un *appel* ou *invocation* s'effectue avec une fonction. Elle a comme résultat l'exécution de la fonction à l'emplacement où cet appel à lieu. Il y a quatre syntaxes :

1. `nomFonction(arguments)` : s'utilise sur les fonctions définies par l'utilisateur, celles importées à travers des modules ou celle définies par l'objet global. Si la fonction appelée existe et se trouve dans la portée de l'appel, le résultat est l'exécution de la fonction. Sinon, JavaScript signale une erreur.
2. `nomFonction.?arguments` : s'utilise sur les fonctions définies par l'utilisateur, celles importées à travers des modules ou celle définies par l'objet global. Si la fonction appelée existe et se trouve dans la portée de l'appel, le résultat est l'exécution de la fonction. Sinon, le résultat est la valeur `undefined`.
3. `nomObjet.nomMethode(arguments)` : s'utilise sur les fonctions contenues dans des objets. Si la fonction appelée existe et se trouve dans la portée de l'appel, le résultat est l'exécution de la fonction, autrement appelée méthode. Si la fonction est hors de portée ou n'existe pas, JavaScript signale une erreur.
4. `nomObjet.?nomMethode(arguments)` : s'utilise sur les fonctions contenues dans des objets. Si la fonction est hors de portée ou n'existe pas, JavaScript signale une erreur. Si la fonction appelée existe et se trouve dans la portée de l'appel, le résultat est l'exécution de la fonction. Sinon, le résultat est la valeur `undefined`.

On en parle en profondeur au chapitre 7.

EXPRESSIONS : EXPRESSIONS DE CREATION D'OBJETS

Une expression de création d'objet à comme résultat la création d'un objet et l'initialisation des propriétés de cet objet. Il y a deux syntaxes : `new nomConstructeur` ou `new nomConstructeur(arguments)`. On utilise cette expression pour créer un objet avec un constructeur. On en parle au chapitre 5.

PAUSE : GENERALITES SUR LES OPERATEURS EN JAVASCRIPT

Tous les types d'expressions qui suivront utilisent des opérateurs. Ces opérateurs à leur tour utilisent des opérandes (il s'agit simplement d'autres expressions) qu'ils évaluent. De par le nombre d'opérandes qu'ils nécessitent, il existe des opérateurs unaires (ils n'ont besoin que d'un seul opérande), binaires (deux) et ternaires (trois).

La plupart des opérateurs acceptent des opérandes de types différents. Avant l'évaluation de ce genre d'expressions, JavaScript peut effectuer une *conversion* implicite (à votre insu). On peut aussi manuellement convertir le type de certaines valeur à notre gré (on utilise généralement des propriétés de l'objet global pour ça, comme la fonction `Number()` par exemple). Cependant, il est important de savoir comment les différents opérandes de nos expressions seront implicitement convertis afin d'appréhender le type du résultat après évaluation de l'expression (par exemple, quel serait le type de la valeur résultante de l'addition d'un nombre avec du texte ?).

L'utilisation de certains opérateurs produit des effets collatéraux (*side effects*) sur vos scripts. Par exemple, `=, +=, -=, %=, &=, *=, **=, /=, ^=, |=, <<=, >>=, >>>=, ++, --` et `delete` ont des effets collatéraux. Ils seront vus chacun à leur tour.

Une expression peut avoir beaucoup d'opérateurs. Et il est de votre devoir de savoir l'ordre selon lequel ces opérateurs seront évalués. Le tableau 4-1 dans le livre : « *JavaScript: The definitive guide* » liste l'ordre de priorité des opérateurs, en partant de ceux avec la plus grande priorité à ceux avec la plus petite.

Par défaut, les expressions en JavaScript sont évaluées de gauche vers la droite (ex : `2+2+2` s'évalue `(2+2)+2`). Cependant, on peut utiliser des parenthèses pour imposer l'ordre que l'on veut.

EXPRESSIONS : EXPRESSIONS ARITHMETIQUES

Une expression arithmétique traduit un calcul mathématique. Elle utilise des opérateurs arithmétiques. Le résultat d'une telle expression est généralement une valeur de type `number`.

1. **L'expression unaire de signe** : cette expression consiste en un opérande et un opérateur « `+` » ou « `-` ». La syntaxe est : `+opérande` ou `-opérande`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions qui s'évaluent en une valeur de type primitif. Cette expression à comme résultat une valeur signée de type `number` (ex : `+"2"` s'évalue à `2`, `-true` s'évalue à `-1`). Elle ne s'applique que sur les valeurs de types primitifs.

2. **L'expression d'addition** : cette expression consiste en deux opérandes et un opérateur « `+` ». La syntaxe est : `opérande1+opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions qui s'évaluent en une valeur de type primitif.

- Lorsque les deux opérandes sont de type `number`, la valeur résultante est aussi de type `number` et consiste en la *somme algébrique* des deux opérandes (ex : `3+2` s'évalue à `5`).
- Lorsque les deux opérandes sont de type `string`, la valeur résultante est aussi de type `string` et consiste en la *concaténation* des deux opérandes (ex : `"2"+"5"` s'évalue à `"25"`).
- Lorsque les deux opérandes sont de type `boolean`, la valeur résultante est de type `number` et consiste en la *somme algébrique* des deux opérandes, `true` étant converti en `1` et `false` étant converti en `0` (ex : `true+true` s'évalue à `2`).
- Lorsque l'un des opérandes est de type `boolean`, cet opérande est converti en `number`, ou en `string`. Le type de la valeur résultante dépend du type de l'autre opérande (ex : `true+2` s'évalue à `3` et `true+"2"` s'évalue à `"true2"`).
- Lorsque les deux opérandes sont `null`, la valeur résultante est de type `number` et vaut `0`.
- Lorsqu'au moins un des opérandes est de type `string`, l'autre opérande est aussi converti en `texte`, puis les deux sont concaténés. Le résultat est de type `string` (ex : `3+"A"` s'évalue à `"3A"`).
- Lorsqu'au moins un des opérandes est `undefined`, Le résultat est `NaN`.
- Lorsque l'un des opérandes est `null`, cet opérande est converti en `0`, ou en `string`. Le type de la valeur résultante dépend du type de l'autre opérande (ex : `null+2` s'évalue à `2` et `null+"2"` s'évalue à `"null2"`).
- Lorsque l'un des opérandes est un `objet` et l'autre est de type `number`, la valeur résultante est l'opérande qui est de type `number` (ex : `{ }+3` s'évalue à `3`).

- Lorsque l'un des opérandes est un objet et l'autre est de type boolean, la valeur résultante est de type number et consiste en 1 pour true et 0 pour false (ex : {}+false s'évalue à 0).
 - Lorsque l'un des opérandes est un objet et l'autre est de type string, le type de la valeur résultante est soit number (ex : {}+"2" s'évalue à 2), soit NaN (ex : {}+"2SD" s'évalue à NaN car il y a des caractères alphabétiques dans le texte).
 - Lorsque l'un des opérandes est un objet, JavaScript essaie de convertir l'objet en un number en appelant les méthodes de conversion de l'objet (valueOf() et toString()). Si la conversion réussit, l'addition est effectuée comme pour les nombres. Sinon, le résultat est NaN (ex : {valueOf: ()=>2}+3 s'évalue à 5, mais {}+2 s'évalue à NaN).
 - Lorsque les deux opérandes sont des objets, JavaScript essaie de convertir chaque objet en un number en appelant les méthodes de conversion de chaque objet. Si les deux conversions réussissent, la division est effectuée comme pour les nombres. Sinon, le résultat est NaN (ex : {}+{} s'évalue à NaN).
3. **L'expression de soustraction** : cette expression consiste en deux opérandes et un opérateur « - ». La syntaxe est : **opérande1 - opérande2**. Où opérande s'agit de n'importe lequel des 12 types d'expressions qui s'évaluent en une valeur de type primitif.
- Lorsque les deux opérandes sont de type number, la valeur résultante est aussi de type number et consiste en la somme algébrique des deux opérandes (ex : 3-2 s'évalue à 1).
 - Lorsque les deux opérandes sont de type boolean, la valeur résultante est de type number et consiste en la somme algébrique des deux opérandes, true étant convertie en 1 et false étant convertie en 0 (ex : true-true s'évalue à 0).
 - Lorsque l'un des opérandes est de type boolean, cet opérande est converti en number, puis s'en suit la conversion jusqu'au type adéquat. Le type de la valeur résultante dépend du type de l'autre opérande (ex : true-2 s'évalue à -1).
 - Lorsque l'un des opérandes est de type string, cet opérande est converti en number, sous-trait comme un nombre. Le type de la valeur résultante est soit number (ex : 3-"2" s'évalue à 1), soit NaN (ex : 3-"2SD" s'évalue à NaN car il y a des caractères alphabétiques dans le texte).
 - Lorsque les deux opérandes sont null, la valeur résultante est de type number et vaut 0.
 - Lorsqu'au moins un des opérandes est undefined, le résultat est NaN.
 - Lorsque les deux opérandes sont des objets, la valeur résultante est NaN.
 - Lorsque l'un des opérandes est un objet et l'autre est de type number, la valeur résultante est l'opérande qui est de type number (ex : {}-3 s'évalue à -3).
 - Lorsque l'un des opérandes est un objet et l'autre est de type boolean, la valeur résultante est de type number et consiste en 1 pour true et 0 pour false (ex : {m:1}-false s'évalue à 0).
 - Lorsque l'un des opérandes est un objet et l'autre est de type string, le type de la valeur résultante est soit number (ex : {}-"2" s'évalue à 2), soit NaN (ex : {}-"2SD" s'évalue à NaN car il y a des caractères alphabétiques dans le texte).
 - Lorsque l'un des opérandes est un objet, JavaScript essaie de convertir l'objet en un number en appelant les méthodes de conversion de l'objet (valueOf() et toString()). Si la conversion réussit, la soustraction est effectuée comme pour les nombres. Sinon, le résultat est NaN (ex : {valueOf: ()=>5}-6 s'évalue à -1, mais {}-2 s'évalue à NaN).
 - Lorsque les deux opérandes sont des objets, JavaScript essaie de convertir chaque objet en un number en appelant les méthodes de conversion de chaque objet. Si les deux conversions réussissent, la soustraction est effectuée comme pour les nombres. Sinon, le résultat est NaN (ex : {}-{} s'évalue à NaN).
4. **L'expression de multiplication** : cette expression consiste en deux opérandes et un opérateur « * ». La syntaxe est : **opérande1*opérande2**. Où opérande s'agit de n'importe lequel des 12 types d'expressions qui s'évaluent en une valeur de type primitif.
- Lorsque les deux opérandes sont de type number, la valeur résultante est aussi de type number et consiste en la multiplication des deux opérandes (ex : 3*2 s'évalue à 6).
 - Lorsque l'un des opérandes est de type number et l'autre est de type string, le type de la valeur résultante est soit number (ex : 2*"2" s'évalue à 4), soit NaN (ex : 8*"2SD" s'évalue à NaN car il y a des caractères alphabétiques dans le texte).

- Lorsque l'un des opérandes est de type number et l'autre est de type boolean, la valeur résultante est de type number et consiste en la *multiplication* des deux opérandes, sachant que `true` est converti en `1` et `false` est converti en `0` (ex : `5 * false` s'évalue à `0`).
- Lorsque l'un des opérandes est un objet, JavaScript essaie de convertir l'objet en un number en appelant les méthodes de conversion de l'objet (`valueOf()` et `toString()`). Si la conversion réussit, la multiplication est effectuée comme pour les nombres. Sinon, le résultat est `NaN` (ex : `{valueOf: ()=>5}*2` s'évalue à `10`, mais `{}*1` s'évalue à `NaN`).
- Lorsque les deux opérandes sont des objets, JavaScript essaie de convertir chaque objet en un number en appelant les méthodes de conversion de chaque objet. Si les deux conversions réussissent, la multiplication est effectuée comme pour les nombres. Sinon, le résultat est `NaN` (ex : `{}*{}` s'évalue à `NaN`).

5. L'expression de division : cette expression consiste en deux opérandes et un opérateur « `/` ». La syntaxe est : `opérande1/opérande2`. Où opérande s'agit de n'importe lequel des 12 types d'expressions qui s'évaluent en une valeur de type primitif.

- Lorsque les deux opérandes sont de type number, la valeur résultante est aussi de type number et consiste en la *division* de l'opérande de gauche par celui de droite. (ex : `5/2` s'évalue à `2.5`).
- Lorsque l'un des opérandes est de type `string`, cet opérande est converti en `number` puis divisé comme un nombre. Le type de la valeur résultante est soit `number` (ex : `9/"3"` s'évalue à `3`), soit `NaN` (ex : `3/"2SD"` s'évalue à `NaN` car il y a des caractères alphabétiques dans le texte).
- Lorsque l'un des opérandes est de type `boolean`, `true` est converti en `1` et `false` est converti en `0`. La division est ensuite effectuée comme pour les nombres (ex : `true/2` s'évalue à `0.5`).
- Lorsque l'un des opérandes est `null`, `null` est converti en `0`. La division est ensuite effectuée comme pour les nombres (ex : `null/2` s'évalue à `0`).
- Lorsque l'un des opérandes est `undefined`, La division avec `undefined` donne toujours `NaN` (ex : `undefined/3` s'évalue à `NaN`).
- Lorsque l'un des opérandes est un objet, JavaScript essaie de convertir l'objet en un number en appelant les méthodes de conversion de l'objet (`valueOf()` et `toString()`). Si la conversion réussit, la division est effectuée comme pour les nombres. Sinon, le résultat est `NaN` (ex : `{valueOf: ()=>2}/3` s'évalue à `8`, mais `{}/2` s'évalue à `NaN`).
- Lorsque les deux opérandes sont des objets, JavaScript essaie de convertir chaque objet en un number en appelant les méthodes de conversion de chaque objet. Si les deux conversions réussissent, la division est effectuée comme pour les nombres. Sinon, le résultat est `NaN` (ex : `{}/{} s'évalue à NaN).`

6. L'expression d'exponentiation : cette expression consiste en deux opérandes et un opérateur « `**` ». La syntaxe est : `opérande1**opérande2`. Où opérande s'agit de n'importe lequel des 12 types d'expressions qui s'évaluent en une valeur de type primitif.

- Lorsque les deux opérandes sont de type number, La valeur résultante est de type number et consiste en l'élévation de l'opérande de gauche à la puissance de l'opérande de droite (ex : `2**3` s'évalue à `8`).
- Lorsque l'un des opérandes est de type `string`, cet opérande est converti en `number` puis divisé comme un nombre. Le type de la valeur résultante est soit `number` (ex : `2**"3"` s'évalue à `8`), soit `NaN` (ex : `3**"2SD"` s'évalue à `NaN` car il y a des caractères alphabétiques dans le texte).
- Lorsque l'un des opérandes est de type `boolean`, `true` est converti en `1` et `false` est converti en `0`. L'exponentiation est ensuite effectuée comme pour les nombres (ex : `true**2` s'évalue à `1`, `false**2` s'évalue à `0`).
- Lorsque l'un des opérandes est `null`, `null` est converti en `0`. L'exponentiation est ensuite effectuée comme pour les nombres (ex : `null**2` s'évalue à `0`).
- Lorsque l'un des opérandes est `undefined`, l'exponentiation avec `undefined` donne toujours `NaN` (ex : `undefined **2` s'évalue à `NaN`).
- Lorsque l'un des opérandes est un objet, JavaScript essaie de convertir l'objet en un number en appelant les méthodes de conversion de l'objet (`valueOf()` et `toString()`). Si la conversion réussit, l'exponentiation est effectuée comme pour les nombres. Sinon, le résultat est `NaN` (ex : `{valueOf: ()=>2}**3` s'évalue à `8`, mais `{}**2` s'évalue à `NaN`).

- Lorsque les deux opérandes sont des objets, JavaScript essaie de convertir chaque objet en un number en appelant les méthodes de conversion de chaque objet. Si les deux conversions réussissent, l'exponentiation est effectuée comme pour les nombres. Si l'une échoue, le résultat est `NaN` (ex : `{valueOf: ()=>2}**{valueOf: ()=>3}` s'évalue à `8`, mais `{ }**{ }` s'évalue à `NaN`).

7. L'expression du modulo

cette expression consiste en deux opérandes et un opérateur « `%` ». La syntaxe est : `opérande1%opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions qui s'évaluent en une valeur de type primitif.

- Lorsque les deux opérandes sont de type `number`, la valeur résultante est de type `number` et consiste en le reste de la division de l'opérande de gauche par celui de droite (ex : `5%2` s'évalue à `1`).
- Lorsque l'un des opérandes est de type `string`, cet opérande est converti en `number` puis divisé comme un nombre. Le type de la valeur résultante est soit `number` (ex : `5%"2"` s'évalue à `1`), soit `NaN` (ex : `5%"2SD"` s'évalue à `NaN` car il y a des caractères alphabétiques dans le texte).
- Lorsque l'un des opérandes est de type `boolean`, `true` est converti en `1` et `false` est converti en `0`. Le modulo est ensuite effectué comme pour les nombres (ex : `true%2` s'évalue à `1`, `false%2` s'évalue à `0`).
- Lorsque l'un des opérandes est `null`, `null` est converti en `0`. Le modulo est ensuite effectué comme pour les nombres (ex : `null%2` s'évalue à `0`).
- Lorsque l'un des opérandes est `undefined`, le modulo avec `undefined` donne toujours `NaN` (ex : `undefined%2` s'évalue à `NaN`).
- Lorsque l'un des opérandes est un objet, JavaScript essaie de convertir l'objet en un `number` en appelant les méthodes de conversion de l'objet (`valueOf()` et `toString()`). Si la conversion réussit, le modulo est effectué comme pour les nombres. Si la conversion échoue, le résultat est `NaN` (ex : `{valueOf: ()=>5}%2` s'évalue à `1`, mais `{ }%2` s'évalue à `NaN`).
- Lorsque les deux opérandes sont des objets, JavaScript essaie de convertir chaque objet en un `number` en appelant les méthodes de conversion de chaque objet. Si les deux conversions réussissent, le modulo est effectué comme pour les nombres. Si l'une des conversions échoue, le résultat est `NaN` (ex : `{valueOf: ()=>5}%{valueOf: ()=>5}` s'évalue à `0`, mais `{ }%{ }` s'évalue à `NaN`).

8. Les expressions bits à bits

Peu couramment utilisés, on en parle pas ici. Voir livre.

EXPRESSIONS : EXPRESSIONS RELATIONNELLES

Une expression relationnelle vérifie la relation entre deux opérandes et retourne une valeur booléenne. Le résultat d'une telle expression est une valeur de type `boolean`.

1. L'expression relationnelle d'égalité non-stricte

cette expression consiste en deux opérandes et un opérateur « `==` ». La syntaxe est : `opérande1==opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si, sans prendre en compte leur type, les valeurs des deux opérandes sont identiques, `false` au cas contraire.

- Lorsque les deux opérandes sont de type `number`, la comparaison se fait directement entre les deux nombres. Si les valeurs sont égales, le résultat est `true`; sinon, c'est `false` (ex : `5==5` s'évalue à `true`, `5==3` s'évalue à `false`).
- Lorsque les deux opérandes sont de type `string`, la comparaison se fait directement entre les deux chaînes de caractères. Si les chaînes sont identiques, le résultat est `true`; sinon, c'est `false` (ex : `"hello"=="hello"` s'évalue à `true`, `"hello"=="world"` s'évalue à `false`).
- Lorsque l'un des opérandes est de type `boolean`, `true` est converti en `1` et `false` est converti en `0`. La comparaison se fait ensuite comme pour les nombres (ex : `true==1` s'évalue à `true`, `false==0` s'évalue à `true`).
- Lorsque l'un des opérandes est `null` ou `undefined`, `null` et `undefined` sont égaux l'un à l'autre mais ne sont égaux à aucune autre valeur (ex : `null==undefined` s'évalue à `true`, `null==0` s'évalue à `false`).
- Lorsque l'un des opérandes est un `number` et l'autre un `string`, la chaîne est convertie en un `number`. Si la conversion réussit, la comparaison se fait entre les nombres. Si la conversion échoue, le résultat est `false` (ex : `5=="5"` s'évalue à `true`, `5=="abc"` s'évalue à `false`).

- Lorsque l'un des opérandes est un number et l'autre un objet, JavaScript essaie de convertir l'objet en un number en appelant ses méthodes de conversion (`valueOf()` et `toString()`). Si la conversion réussit, la comparaison se fait entre les nombres. Si la conversion échoue, le résultat est `false` (ex : `5=={valueOf:()=>5}` s'évalue à `true`, `5=={} s'évalue à false`).
 - Lorsque les deux opérandes sont des objets, les objets sont comparés par référence, non par valeur. Deux objets sont égaux si et seulement si ils font référence au même objet (ex : `{a:1}=={a:1}` s'évalue à `false` car ce sont deux objets distincts, mais `let obj = {a:1}; obj==obj` s'évalue à `true`).
2. **L'expression relationnelle d'inégalité non-stricte** : cette expression consiste en deux opérandes et un opérateur « `!=` ». La syntaxe est : `opérande1!=opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si, sans prendre en compte leur type, les valeurs des deux opérandes sont différentes, `false` au cas contraire.
- Lorsque les deux opérandes sont de type number, la comparaison se fait directement entre les deux nombres. Si les valeurs sont différentes, le résultat est `true`; sinon, c'est `false` (ex : `5!=3` s'évalue à `true`, `5!=5` s'évalue à `false`).
 - Lorsque les deux opérandes sont de type string, la comparaison se fait directement entre les deux chaînes de caractères. Si les chaînes sont différentes, le résultat est `true`; sinon, c'est `false` (ex : `"hello"!="world"` s'évalue à `true`, `"hello"!="hello"` s'évalue à `false`).
 - Lorsque l'un des opérandes est de type boolean, `true` est converti en `1` et `false` est converti en `0`. La comparaison se fait ensuite comme pour les nombres (ex : `true!=1` s'évalue à `false`, `false!=1` s'évalue à `true`).
 - Lorsque l'un des opérandes est `null` ou `undefined`, `null` et `undefined` sont égaux l'un à l'autre mais ne sont égaux à aucune autre valeur. Ainsi, `null` et `undefined` ne sont pas inégaux entre eux (ex : `null!=undefined` s'évalue à `false`, `null!=0` s'évalue à `true`).
 - Lorsque l'un des opérandes est un number et l'autre un string, la chaîne est convertie en un number. Si la conversion réussit, la comparaison se fait entre les nombres. Si la conversion échoue, le résultat est `true` (ex : `5!="3"` s'évalue à `true`, `5!="5"` s'évalue à `false`).
 - Lorsque l'un des opérandes est un number et l'autre un objet, JavaScript essaie de convertir l'objet en un number en appelant ses méthodes de conversion (`valueOf()` et `toString()`). Si la conversion réussit, la comparaison se fait entre les nombres. Si la conversion échoue, le résultat est `true` (ex : `5!={valueOf:()=>3}` s'évalue à `true`, `5!={valueOf:()=>5}` s'évalue à `false`).
 - Lorsque les deux opérandes sont des objets, les objets sont comparés par référence, non par valeur. Deux objets sont inégaux sauf s'ils font référence au même objet (ex : `{a:1}!={a:1}` s'évalue à `true` car ce sont deux objets distincts, mais `let obj = {a:1}; obj!=obj` s'évalue à `false`).
3. **L'expression relationnelle d'égalité stricte** : cette expression consiste en deux opérandes et un opérateur « `==` ». La syntaxe est : `opérande1==opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si les valeurs des deux opérandes sont identiques et de même type, `false` au cas contraire.
- Lorsque les deux opérandes sont de type number, la comparaison se fait directement entre les deux nombres. Si les valeurs sont identiques, le résultat est `true`; sinon, c'est `false` (ex : `5==5` s'évalue à `true`, `5==3` s'évalue à `false`).
 - Lorsque les deux opérandes sont de type string, la comparaison se fait directement entre les deux chaînes. Si elles sont identiques, le résultat est `true`; sinon, c'est `false` (ex : `"hello"=="hello"` s'évalue à `true`, `"hello"=="world"` s'évalue à `false`).
 - Lorsque les deux opérandes sont de type boolean, la comparaison se fait directement entre les deux valeurs booléennes. Si les valeurs sont identiques, le résultat est `true`; sinon, c'est `false` (ex : `true==true` s'évalue à `true`, `true==false` s'évalue à `false`).
 - Lorsque l'un des opérandes est `null` ou `undefined`, `null` n'est strictement égal qu'à `null`, et `undefined` n'est strictement égal qu'à `undefined`. (ex : `undefined==undefined` s'évalue à `true`, `null==null` s'évalue à `true`, `null==undefined` s'évalue à `false`).
 - Lorsque l'un des opérandes est de type number et l'autre est un string, boolean, `null`, ou `undefined`, il n'y a pas de conversion de type. La comparaison est toujours `false` (ex : `5=="5"` s'évalue à `false`, `1==true` s'évalue à `false`).
 - Lorsque les deux opérandes sont des objets, les objets sont comparés par référence, non par valeur. Deux objets sont égaux seulement s'ils font référence au même objet (ex :

`{a:1}==={a:1}` s'évalue à `false` car ce sont deux objets distincts, mais `let obj = {a:1}; obj==obj` s'évalue à `true`.

4. **L'expression relationnelle d'inégalité stricte** : cette expression consiste en deux opérandes et un opérateur « `!==` ». La syntaxe est : `opérande1 != opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `false` si les valeurs des deux opérandes sont identiques et de même type, `true` au cas contraire.

- Lorsque les deux opérandes sont de type `number`, la comparaison se fait directement entre les deux. Si les valeurs sont différentes, le résultat est `true`; sinon, c'est `false` (ex : `5!==3` s'évalue à `true`, `5!==5` s'évalue à `false`).
- Lorsque les deux opérandes sont de type `string`, la comparaison se fait directement entre les deux chaînes. Si elles sont différentes, le résultat est `true`; sinon, le résultat est `false` (ex : `"hello"!="world"` s'évalue à `true`, `"hello"!="hello"` s'évalue à `false`).
- Lorsque les deux opérandes sont de type `boolean`, la comparaison se fait directement entre les deux valeurs booléennes. Si les valeurs sont différentes, le résultat est `true`; sinon, c'est `false` (ex : `true!=false` s'évalue à `true`, `true==true` s'évalue à `false`).
- Lorsque les deux opérandes sont des objets, ils sont comparés par référence. Deux objets sont strictement inégaux s'ils ne font pas référence au même objet (ex : `{a:1}!=={a:1}` s'évalue à `true` car ce sont deux objets distincts, mais `let obj={a:1}; obj==obj` s'évalue à `false`).
- Lorsque les opérandes sont de types différents, la comparaison strictement inégale est toujours `true` (ex : `5!="5"` s'évalue à `true`, `true!=1` s'évalue à `true`).

5. **L'expression relationnelle « strictement inférieur »** : cette expression consiste en deux opérandes et un opérateur « `<` ». La syntaxe est : `opérande1<opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si la valeur de l'opérande de gauche n'est ni supérieure, ni égale à celle de l'opérande de droite. Elle retourne `false` au cas contraire.

- Lorsque les deux opérandes sont de type `number`, la comparaison se fait directement entre les deux nombres. Si la valeur de gauche est strictement inférieure à la valeur de droite, le résultat est `true`; sinon, c'est `false` (ex : `3<5` s'évalue à `true`, `5<3` s'évalue à `false`).
- Lorsque les deux opérandes sont de type `string`, les chaînes de caractères sont comparées selon l'ordre lexicographique basé sur les valeurs Unicode de chaque caractère. Si la chaîne de gauche est strictement inférieure à la chaîne de droite, le résultat est `true`; sinon, c'est `false` (ex : `"apple"<"banana"` s'évalue à `true`, `"banana"<"apple"` s'évalue à `false`).
- Lorsque les deux opérandes sont de type `boolean`, les valeurs `true` et `false` sont converties en `1` et `0` respectivement, la comparaison se fait comme pour les nombres (ex : `false<true` s'évalue à `true`, `true<false` s'évalue à `false`).
- Lorsque l'un des opérandes est `number` et l'autre `string`, `string` est convertie en nombre avant la comparaison (ex : `5<"10"` s'évalue à `true`, `"10"<5` s'évalue à `false`).
- Lorsque l'un des opérandes est `boolean`, le booléen est converti en nombre avant la comparaison (`true` devient `1` et `false` devient `0`) (ex : `true<2` s'évalue à `true`, `false<1` s'évalue à `true`).
- Lorsque l'un des opérandes est un objet, JavaScript le convertit en sa valeur primitive en utilisant la méthode `valueOf()` ou `toString()`, selon les cas, avant la comparaison (ex : `{valueOf: ()=>5}<10` s'évalue à `true`).

6. **L'expression relationnelle « inférieur ou égal »** : cette expression consiste en deux opérandes et un opérateur « `<=` ». La syntaxe est : `opérande1<=opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si la valeur de l'opérande de gauche est inférieure ou égale à celle de l'opérande de droite. Elle retourne `false` au cas contraire.

- Lorsque les deux opérandes sont de type `number`, La comparaison se fait directement entre les deux nombres. Si la valeur de gauche est inférieure ou égale à la valeur de droite, le résultat est `true`; sinon, c'est `false` (ex : `3<=5` s'évalue à `true`, `5<=3` s'évalue à `false`, `5<=5` s'évalue à `true`).
- Lorsque les deux opérandes sont de type `string`, les chaînes de caractères sont comparées selon l'ordre lexicographique basé sur les valeurs Unicode de chaque caractère. Si la chaîne de gauche est inférieure ou égale à la chaîne de droite, le résultat est `true`; sinon,

c'est `false` (ex : `"a" <= "b"` s'évalue à `true`, `"b" <= "a"` s'évalue à `false`, `"apple" <= "apple"` s'évalue à `true`).

- Lorsque les deux opérandes sont de type boolean, les valeurs `true` et `false` sont converties en `1` et `0` respectivement, la comparaison se fait comme pour les nombres (ex : `false <= true` s'évalue à `true`, `true <= false` s'évalue à `false`, `true <= true` s'évalue à `true`).
- Lorsque l'un des opérandes est number et l'autre string, la chaîne de caractères est convertie en nombre avant la comparaison (ex : `5 <= "10"` s'évalue à `true`, `"10" <= 5` s'évalue à `false`, `"5" <= 5` s'évalue à `true`).
- Lorsque l'un des opérandes est boolean, le booléen est converti en nombre avant la comparaison (`true` devient `1` et `false` devient `0`) (ex : `true <= 2` s'évalue à `true`, `false <= 1` s'évalue à `true`, `true <= 1` s'évalue à `true`).
- Lorsque l'un des opérandes est un objet, l'objet est converti en sa valeur primitive en utilisant la méthode `valueOf()` ou `toString()`, selon les cas, avant la comparaison (ex : `{valueOf: () => 5} <= 10` s'évalue à `true`, `{toString: () => "5"} <= 10` s'évalue à `true`).

7. L'expression relationnelle « strictement supérieur » : cette expression consiste en deux opérandes et un opérateur « `>` ». La syntaxe est : `opérande1 > opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si la valeur de l'opérande de gauche n'est ni inférieure, ni égale à celle de l'opérande de droite. Elle retourne `false` au cas contraire.

- Lorsque les deux opérandes sont de type number, la comparaison se fait directement entre les deux nombres. Si la valeur de gauche est strictement supérieure à la valeur de droite, le résultat est `true`; sinon, c'est `false` (ex : `5 > 3` s'évalue à `true`, `3 > 5` s'évalue à `false`).
- Lorsque les deux opérandes sont de type string, les chaînes de caractères sont comparées selon l'ordre lexicographique basé sur les valeurs Unicode de chaque caractère. Si la chaîne de gauche est strictement supérieure à la chaîne de droite, le résultat est `true`; sinon, c'est `false` (ex : `"banana" > "apple"` s'évalue à `true`, `"apple" > "banana"` s'évalue à `false`).
- Lorsque les deux opérandes sont de type boolean, les valeurs `true` et `false` sont converties en `1` et `0` respectivement, et la comparaison se fait comme pour les nombres (ex : `true > false` s'évalue à `true`, `false > true` s'évalue à `false`).
- Lorsque l'un des opérandes est number et l'autre string, la chaîne de caractères est convertie en nombre avant la comparaison (ex : `10 > "5"` s'évalue à `true`, `"5" > 10` s'évalue à `false`).
- Lorsque l'un des opérandes est boolean, le booléen est converti en nombre avant la comparaison (`true` devient `1` et `false` devient `0`) (ex : `2 > true` s'évalue à `true`, `1 > false` s'évalue à `true`).
- Lorsque l'un des opérandes est un objet, l'objet est converti en sa valeur primitive en utilisant la méthode `valueOf()` ou `toString()`, selon les cas, avant la comparaison (ex : `{valueOf: () => 10} > 5` s'évalue à `true`).

8. L'expression relationnelle « supérieur ou égal » : cette expression consiste en deux opérandes et un opérateur « `>=` ». La syntaxe est : `opérande1 >= opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si la valeur de l'opérande de gauche est supérieure ou égale à celle de l'opérande de droite. Elle retourne `false` au cas contraire.

- Lorsque les deux opérandes sont de type number, la comparaison se fait directement entre les deux nombres. Si la valeur de gauche est supérieure ou égale à la valeur de droite, le résultat est `true`; sinon, c'est `false` (ex : `5 >= 3` s'évalue à `true`, `3 >= 5` s'évalue à `false`, `5 >= 5` s'évalue à `true`).
- Lorsque les deux opérandes sont de type string, les chaînes de caractères sont comparées selon l'ordre lexicographique basé sur les valeurs Unicode de chaque caractère. Si la chaîne de gauche est supérieure ou égale à la chaîne de droite, le résultat est `true`; sinon, c'est `false` (ex : `"b" >= "a"` s'évalue à `true`, `"a" >= "b"` s'évalue à `false`, `"apple" >= "apple"` s'évalue à `true`).
- Lorsque les deux opérandes sont de type boolean, les valeurs `true` et `false` sont converties en `1` et `0` respectivement, et la comparaison se fait comme pour les nombres (ex : `true >= false` s'évalue à `true`, `false >= true` s'évalue à `false`, `true >= true` s'évalue à `true`).

- Lorsque l'un des opérandes est `number` et l'autre `string`, la chaîne de caractères est convertie en nombre avant la comparaison (ex : `10>="5"` s'évalue à `true`, `"5">=10` s'évalue à `false`, `5>="5"` s'évalue à `true`).
- Lorsque l'un des opérandes est `boolean`, le booléen est converti en nombre avant la comparaison (`true` devient `1` et `false` devient `0`) (ex : `2>=true` s'évalue à `true`, `1>=false` s'évalue à `true`, `1>=true` s'évalue à `true`).
- Lorsque l'un des opérandes est un objet, l'objet est converti en sa valeur primitive en utilisant la méthode `valueOf()` ou `toString()`, selon les cas, avant la comparaison (ex : `{valueOf: ()=>10}>=5` s'évalue à `true`, `{toString: ()=>"10"}>=5` s'évalue à `true`).

9. **L'expression relationnelle `in`** : cette expression consiste en deux opérandes et un opérateur « `in` ». La syntaxe est : `opérande in objet`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si la valeur de l'opérande correspond au nom d'une propriété de `objet`. Elle retourne `false` au cas contraire. (ex :

Utilisation avec des objets :

```
const car = {make: 'Toyota', model: 'Corolla', year: 2020};
'make' in car; //s'évalue à true
'color' in car; //s'évalue à false
'toString' in car; //s'évalue à true (hérité du prototype Object)
```

Utilisation avec des tableaux :

```
const fruits = ['apple', 'banana', 'cherry'];
0 in fruits; //s'évalue à true (l'indice 0 existe)
3 in fruits; //s'évalue à false (l'indice 3 n'existe pas)
'length' in fruits; //s'évalue à true (la propriété 'length' existe)
```

L'opérateur `in` n'est pas utilisé directement pour les chaînes de caractères, mais pour vérifier des propriétés existantes sur l'objet `string` :

```
const str = 'hello';
'length' in str; //s'évalue à true (la propriété 'length' existe)
0 in str; //s'évalue à true (l'indice 0 existe)
5 in str; //s'évalue à false (l'indice 5 n'existe pas)
```

Utilisation pour les objets avec héritage

```
function Person(name) {this.name = name;}
Person.prototype.age = 30;
const person1 = new Person('Alice');
'name' in person1; //s'évalue à true
'age' in person1; //s'évalue à true (hérité du prototype Person)).
```

10. **L'expression relationnelle `instanceof`** : cette expression consiste en deux opérandes et un opérateur « `instanceof` ». La syntaxe est : `objet instanceof fonction`. Le résultat de l'expression est `true` si `objet` est une instance d'une classe ou si elle se trouve dans sa chaîne de `prototypes`. Elle retourne `false` au cas contraire (ex :

```
class Animal {}
class Dog extends Animal {}
const myDog = new Dog();
myDog instanceof Dog; //true
myDog instanceof Animal; //true
myDog instanceof Object; //true
myDog instanceof Array; //false).
```

EXPRESSIONS : EXPRESSIONS LOGIQUES

Une expression logique est souvent utilisée avec des expressions relationnelles pour effectuer de l'algèbre booléenne (qui consiste en l'addition, la multiplication ou la négation d'une valeur booléenne). Le résultat d'une telle expression est une valeur de type `boolean`, `true` ou `false`.

1. **L'expression logique et-logique** : cette expression consiste en deux opérandes et un opérateur « `&&` ». La syntaxe est : `opérande1&&opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `true` si la valeur des deux opérandes s'évalue à `true`. Elle retourne `false` au cas contraire.
- Lorsque les deux opérandes sont de type `boolean`, si les deux opérandes sont `true`, l'opérateur retourne `true`. Sinon, il retourne `false`.

- Lorsqu'utilisé seul, comme argument d'une fonction, comme valeur d'un objet ou comme opérande de droite d'une expression d'affectation, l'expression logique produit un effet collatéral qu'on utilise généralement comme court-circuit en JavaScript : l'opérateur `&&` évalue les opérandes de gauche à droite. Si le premier opérande est `false`, il évalue le premier opérande tel quel sans le convertir en boolean et n'évalue pas le deuxième opérande. Si le premier opérande est `true`, il évalue le deuxième opérande tel quel sans le convertir en boolean et n'évalue pas le premier opérande. Ce court-circuit est un sucre syntaxique qui permet d'exécuter du code conditionnellement sans écrire des `if/else` (ex :

Au lieu d'écrire

```
if(a){
    b; //b est executé uniquement si a s'évalue à true
} else {a;} //Sinon, a est executé
```

, on peut simplement écrire

`a&&b; /*si a est évalué à true, b est exécuté. Comme l'expression est utilisée seule, rien d'autre ne se produit à part cet effet collatéral. Si a est évalué à false, rien ne se produit et il n'y a aucun effet collatéral*/`

Aussi,

```
let truc = a&&b; /*si a est évalué à true, b est affecté tel quel à truc. Si a est évalué à false, c'est a qui est affecté à truc*/
console.log(true&&4); //utilisé comme argument. Affiche 4
let obj = {montruc: true&&2}; //la valeur 2 est stockée dans montruc).
```

- Lorsqu'un opérande a comme valeur `false, 0, -0, 0n, '', null, undefined` ou `NaN`, il est évalué à `false`. Dans n'importe quel autre cas, il est évalué à `true`.

2. L'expression logique ou-logique

: cette expression consiste en deux opérandes et un opérateur « `||` ». La syntaxe est : `opérande1 || opérande2`. Où opérande s'agit de n'importe lequel des 12 types d'expressions. Le résultat de l'expression est `false` si la valeur des deux opérandes s'évalue à `false`. Elle retourne `true` au cas contraire.

- Lorsque les deux opérandes sont de type boolean, si les deux opérandes sont `false`, l'opérateur retourne `false`. Sinon, il retourne `true`.
- Lorsqu'utilisé seul, comme argument d'une fonction, comme valeur d'un objet ou comme opérande de droite d'une expression d'affectation, l'expression logique produit un effet collatéral qu'on utilise généralement comme court-circuit en JavaScript : l'opérateur `||` évalue les opérandes de gauche à droite. Si le premier opérande est `true`, il évalue le premier opérande tel quel sans le convertir en boolean et n'évalue pas le deuxième opérande. Si le premier opérande est `false`, il évalue le deuxième opérande tel quel sans le convertir en boolean et n'évalue pas le premier opérande. Ce court-circuit est un sucre syntaxique qui permet d'exécuter du code conditionnellement sans écrire des `if/else` (ex : Au lieu d'écrire

```
if(!a){
    b; //b est executé uniquement si a s'évalue à false
} else {a;} //Sinon, a est executé
```

, on peut simplement écrire

`a||b; /*si a est évalué à false, b est exécuté. Comme l'expression est utilisée seule, rien d'autre ne se produit à part cet effet collatéral. Si a est évalué à true, rien ne se produit et il n'y a aucun effet collatéral*/`

Aussi,

```
let truc = a||b; /*si a est évalué à false, b est affecté tel quel à truc. Si a est évalué à true, c'est a qui est affecté à truc*/
console.log(false||4); //utilisé comme argument. Affiche 4
let obj = {montruc: false||2}; //la valeur 2 est stockée dans montruc).
```

- Lorsque un opérande a comme valeur `false, 0, -0, 0n, '', null, undefined` ou `NaN`, il est évalué à `false`. Dans n'importe quel autre cas, il est évalué à `true`.

3. L'expression logique non-logique

: cette expression consiste en un opérande et un opérateur « `!` ». La syntaxe est : `!opérande`. Où opérande s'agit de n'importe lequel des 12 types d'expressions.

- Lorsque l'opérande est une valeur de type boolean, si l'opérande est `true`, l'opérateur retourne `false`. Si l'opérande est `false`, l'opérateur retourne `true`.
- Lorsque l'opérande n'est pas une valeur booléenne, JavaScript convertit l'opérande en une valeur booléenne avant d'appliquer l'opérateur `!`. Les valeurs `falsy` (comme `0, -0, 0n, '', null,`

`undefined` ou `NaN`) sont converties en `false`, puis inversées pour retourner `true`. Les valeurs `truthy` (toutes les valeurs qui ne sont pas `falsy`) sont converties en `true`, puis inversées pour retourner `false` (ex : `!"10"` s'évalue à `false`).

EXPRESSIONS : EXPRESSIONS D'ASSIGNATION

Une expression d'assignation est utilisée pour assigner une valeur à une variable ou propriété. Cette expression ne produit aucun résultat. Il est plutôt utilisé pour son effet collatéral : une fois une valeur assignée à une variable en utilisant cette expression, toutes les références à cette variable seront évaluées en fonction de cette valeur. On l'utilise pour initialiser ou définir une variable et pour définir la valeur de la propriété d'un objet ou de l'élément d'un tableau.

1. **L'expression d'assignation directe** : cette expression consiste en deux opérandes et un opérateur « `=` ». La syntaxe est : `opérande1=opérande2`. Où `opérande1` consiste en l'identifiant d'une variable, une expression d'accès à la propriété d'un objet ou une expression d'accès à la propriété d'un tableau et `opérande2` s'agit de n'importe lequel des 12 types d'expressions (ex : `nom='guy'` ;).
2. **L'expression d'assignation avec opération** : cette expression consiste en deux opérandes et un opérateur « `op=` ». La syntaxe est : `opérande1op=opérande2`. Où `opérande1` consiste en l'identifiant d'une variable, une expression d'accès à la propriété d'un objet ou une expression d'accès à la propriété d'un tableau, `opérande2` s'agit de n'importe lequel des 12 types d'expressions et `op` s'agit d'un des opérateurs `+, -, %, &, *, **, /, ^, |, <<, >>` et `>>>`. C'est un sucre syntaxique qui équivaut à écrire `opérande1=opérande1opopérande2` (ex : `nom+= 'guy'` ; équivaut à écrire `nom=nom+ 'guy'` ; la nouvelle valeur stockée dans `nom` est `'guyguy'`).
3. **L'expression d'incrémation** : cette expression est une forme spéciale de l'assignation avec opération. Elle consiste en un opérande et un opérateur « `++` » et consiste en l'incrémation de la valeur de l'opérande par 1. Il y a 2 syntaxes : `opérande++` et `++opérande`. Où `opérande` consiste en l'identifiant d'une variable, une expression d'accès à la propriété d'un objet ou une expression d'accès à la propriété d'un tableau. Utilisé seuls, comme par exemple comme incrément d'une boucle `for` ou `while`, il n'y a aucune différence entre les deux (du fait qu'il n'y a aucun effet collatéral). Mais lorsqu'il faut assigner cette expression à une autre variable, chaque syntaxe produit un effet collatéral différent (ex : `a=b++`; s'évalue comme suit : `a=b`; `b=b+1`; et `a=++b`; s'évalue comme suit : `a=b+1`; `b=b+1`;).
4. **L'expression de décrémation** : cette expression est une forme spéciale de l'assignation avec opération. Elle consiste en un opérande et un opérateur « `--` » et consiste à la décrémation de la valeur de l'opérande par 1. Il y a 2 syntaxes : `opérande--` et `--opérande`. Où `opérande` consiste en l'identifiant d'une variable, une expression d'accès à la propriété d'un objet ou une expression d'accès à la propriété d'un tableau. Utilisé seuls, comme par exemple comme incrément d'une boucle `for` ou `while`, il n'y a aucune différence entre les deux (du fait qu'il n'y a aucun effet collatéral). Mais lorsqu'il faut assigner cette expression à une autre variable, chaque syntaxe produit un effet collatéral différent (ex : `a=b--`; s'évalue comme suit : `a=b`; `b=b-1`; et `a=--b`; s'évalue comme suit : `a=b-1`; `b=b-1`;).

EXPRESSIONS : EXPRESSIONS D'EVALUATION

Une expression d'évaluation consiste en une fonction de l'objet global (qui deviendra dans le futur un opérateur) : `eval()`. Cette expression prend en paramètre une chaîne de caractères contenant une expression parmi les 12 types et renvoie le résultat de l'évaluation (ex : `eval("10+5")` s'évalue à `15`).

EXPRESSIONS : EXPRESSIONS DIVERSES

Il s'agit d'une expression qui n'a pas de classification précise. Elle peut n'être qu'un sucre syntaxique, ou n'être utilisé que pour son effet collatéral, etc.

1. **L'expression « `? :` »** : Il s'agit d'un sucre syntaxique à l'instruction `if/else`. Au lieu d'écrire :


```
if(expr1){ //Si expr1 est évaluée à true
    expr2; //expr2 est évaluée
} else{
    expr3; //Sinon, c'est expr3 qui est évaluée
}
```

- , on écrit : `expr1 ? expr2 : expr3`. Où `expr1` est une expression relationnelle et/ou logique ; `expr2` et `expr3` s'agit d'une des 12 types d'expressions (ex : `const valAbs = (x>0) ? x : -x;`).
2. **L'expression « ?? »** : La syntaxe est : `opérande1??opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Elle évalue `opérande1`, si elle vaut `undefined` ou `null`, elle évalue `opérande1` tel quel. Sinon, elle évalue `opérande2` tel quel. Il s'agit d'un sucre syntaxique à l'expression suivante : `(a!==null)&&(a!==undefined) ? a : b;`. Au lieu d'écrire cette dernière, on écrit juste : `a??b;`. Lorsqu'utilisé dans des expressions qui ont des effets collatéraux, ils produisent les mêmes effets que les court-circuits avec les expressions logiques `&&` et `||` (ex :
`let x = a??b; //Si a==null ou a==undefined, x=a, sinon, x=b).`
 3. **L'expression `typeof`** : La syntaxe est : `typeof opérande`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Cette expression a comme résultat, une chaîne de caractère contenant le nom du type du résultat de `opérande` (ex : `typeof x>0` s'évalue à "boolean").
 4. **L'expression `delete`** : La syntaxe est : `delete opérande`. Où `opérande` s'agit d'une expression d'accès à la propriété d'un objet ou à l'élément d'un tableau. Cette expression n'a rien comme résultat. Elle s'utilise uniquement pour son effet collatéral : elle permet de supprimer la propriété de l'objet ou l'élément du tableau qui lui est passé comme opérande (ex :
`let a = {salut:"coucou", nom:"chris"}; //Un objet avec deux propriétés
let b = [1,3]; //Un tableau avec deux éléments
delete a.nom; //On a supprimé La propriété nom, on ne peut plus l'utiliser
delete a[1]; //On a supprimé le deuxième élément, on ne peut plus l'utiliser).`
 5. **L'expression `await`** : Utile pour la programmation asynchrone. Voir chapitre 12.
 6. **L'expression `void`** : La syntaxe est : `void opérande`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Cette expression a comme résultat, `undefined`. Elle s'utilise rarement et uniquement lorsqu'on veut utiliser une expression uniquement pour son effet collatéral (ex :
`let a = 1, b = 5;
b = void a++;
a; //est incrémenté à 2, par effet collatéral
b; //contient la valeur undefined du fait qu'on a utilisé void).`
 7. **L'expression « , »** : La syntaxe est : `opérande1,opérande2`. Où `opérande` s'agit de n'importe lequel des 12 types d'expressions. Cette expression évalue `opérande1`, puis `opérande2` et a comme résultat la valeur de `opérande2`. Elle n'est utilisée que pour son effet collatéral : Elle évalue deux opérandes l'une après l'autre. C'est un sucre syntaxique qui nous permet d'écrire une seule instruction, mais qui se comporte comme deux (ex : Au lieu d'écrire,
`let x = 13;
let y = 9;`
on peut simplement écrire : `let x = 13, y = 9;`
ou : `void x = 13, y = 9; //Crée implicitement deux variables avec le mot clé var).`

Enfin, toutes ces expressions peuvent être mélangées à votre gré pour créer des expressions complexes.

CHAPITRE 4 : INSTRUCTIONS

Les scripts JavaScript sont exécutés instruction par instruction. Une instruction est un ordre qu'on donne à l'interpréteur JavaScript lui disant d'évaluer une expression, exécuter une structure de contrôle, une boucle, etc.). Toute instruction qui n'est pas un bloc se termine par un point-virgule (`;`). Même si JavaScript rajoute automatiquement des points-virgules là où il estime qu'il en manque, il est recommandé de toujours manuellement terminer chaque instruction par un point-virgule.

INSTRUCTIONS SIMPLES

Toutes les expressions qui produisent des effets collatéraux sont des instructions simples : expressions d'assignation, expressions diverses, expressions d'appel de fonction, etc.

INSTRUCTIONS : INSTRUCTIONS COMPOSÉES

Une instruction composée, appelée aussi « *bloc d'instructions* », est une liste d'instructions se trouvant entre deux accolades (`{}`) et séparées par des points-virgules (`;`) (ex :

```
{
  //Un bloc avec deux instructions simples
  let a=3;
  console.log(a);
}).
```

On utilise les instructions composées pour 4 raisons :

1. Pour exécuter une liste d'instructions de manière ordonnée. C'est souvent le cas dans les structures de contrôle.
2. Pour gérer la portée de certaines variables qu'on veut créer. Une variable déclarée dans un bloc n'est utilisable que dans ce bloc.
3. Pour augmenter la lisibilité de son code.
4. Pour éviter de polluer la portée globale avec des variables temporaires.

Les blocs ne se terminent pas par des points-virgules (mais les instructions qui s'y trouvent, si).

INSTRUCTIONS VIDES

Une instruction vide s'agit d'une expression qui, utilisée seule, n'a aucun effet collatéral ou d'un point-virgule utilisé seul (`;`). On en utilise lorsqu'on veut créer une structure de contrôle sans corps.

STRUCTURE DE CONTROLE : INSTRUCTIONS CONDITIONNELLES

Une instruction conditionnelle s'agit d'une instruction ou d'un bloc d'instruction qui n'est exécutée que si une condition spécifiée au préalable est remplie. Il y a trois types d'instructions conditionnelles :

1. L'instruction `if` : La syntaxe est : `if(expression) instruction`; ou `if(expression) bloc`.

Si `expression` est évaluée à `true`, l'instruction ou le bloc est exécutée. Sinon, elle est ignorée (ex :
`//Si x<7 est évalué à true, l'instruction est exécutée`
`if(x<7) console.log('coucou');`

`//Si a est égal à 3, ce bloc est exécuté`
`if(a==3){`
 `let nom = "Paul";`
 `console.log(`hello ${nom}`);`
`}.`

2. L'instruction `if/else` : La syntaxe est : `if(expression) instruction1; else instruction2;`
 Ou `if(expression) bloc1 else bloc2.`

Si `expression` est évaluée à `true`, `instruction1` ou `bloc1` est exécutée. Sinon, c'est `instruction2` ou `bloc2` qui est exécutée (ex :

`//en utilisant des instructions simples`
`if(n==1) console.log("Tu as 1 nouveau message.");`
`else console.log(`tu as ${n} nouveaux messages.');`

`//en utilisant des blocs`
`if(i==j){`
 `console.log("i equals k");`
`} else{`
 `console.log("i doesn't equal j");`
`}.`

`//On peut aussi mettre d'autres if comme instructions dans les else`
`if(n==1){`
 `//Execute code bloc1`
`} else if(n==2){`
 `//Execute code bloc2`
`} else if(n==3){`
 `//Execute code bloc3`
`} else{`
 `//Si toutes ces conditions échouent, exécute bloc4`
`}.`

3. L'instruction `switch` : La syntaxe est :

Cas général

```
switch(expression){
    case valeur1 :
        Instructions1
        break;
    case valeur2 :
        Instructions2
        break;
    //...Ainsi de suite
    default :
        InstructionsN
        break;
}
```

Dans une fonction, on peut utiliser aussi `return`

```
switch(expression){
    case valeur1 :
        Instructions1
        return expr1;
    case valeur2 :
        Instructions2
        return expr2;
    //...Ainsi de suite
    default :
        InstructionsN
        return exprN;
}
```

Où `expression` et `expr` s'agissent de n'importe lequel des 12 types d'expressions. Si, après évaluation, `expression==valeur1` est évaluée à `true`, `Instructions1` sont exécutés (`break` ou `return` permet de quitter le `switch`). Si `expression==valeur2` est évaluée à `true`, `Instructions2` sont exécutés (`break` ou `return` permet de quitter le `switch`). Ainsi de suite. Si `expression` n'est évalué à `true` pour aucune des valeurs spécifiées, `InstructionsN` sont exécutés (ex :

```
switch(x){
    case 1 : //Si x==1, exécuter :
        console.log('x vaut 1');
        console.log('tu comprends ?');
        break;
    case 2 : //Si x==2, exécuter :
        console.log('x vaut 2');
        break;
    default : //Si x ne vaut aucune valeur spécifiée, exécuter :
        break;
}).
```

STRUCTURE DE CONTROLE : BOUCLES

Une boucle s'agit d'une instruction ou d'un bloc d'instruction qui est exécutée de manière répétitive tant qu'une condition spécifiée au préalable n'est pas remplie. Il y a cinq types de boucles :

1. La boucle `while` : La syntaxe est : `while(expression) instruction;`
ou `while(expression) bloc.`

Si `expression` est évaluée à `true`, l'instruction ou le bloc est exécutée de manière répétitive jusqu'à ce qu'un facteur interne (incrément, `break`, `continue` ou `return`) lui rende `false`. (ex :

```
while(x<10) x++; //Lorsque x = 10, la boucle s'arrête
while(true){
    console.log('coucou');
    break; //break permet de sortir de la boucle
}
let a = ()=>{
    while(!false){
        let rep = console.log('aaaa');
        return rep; //permet de sortir de la boucle utilisée dans une fonction
    }
};
```

2. La boucle `do/while` : La syntaxe est : `do instruction; while(expression);`
ou `do bloc while(expression);`.

Elle exécute l'instruction ou le bloc une fois, puis elle évalue `expression`. Si elle est évaluée à `true`, l'instruction ou le bloc est re-exécutée de manière répétitive jusqu'à ce qu'un facteur interne (incrément, `break`, `continue` ou `return`) lui rende `false`. On l'utilise pour s'assurer que le contenu d'une boucle est exécutée au moins une fois, peu importe la condition (ex :

```
do console.log('hey');
while(true);
```

```
do{
    i++;
}
while(i<5);
```

3. La boucle **for** : La syntaxe est : **for(exprInit; exprTest; exprIncr) instruction;**
ou **for(exprInit; exprTest; exprIncr) bloc.**

Elle initialise une ou plusieurs variables **exprInit**, puis elle évalue **exprTest**. S'il est évaluée à **true**, l'instruction ou le bloc est exécutée. A la fin de chaque exécution, les variables sont incrémentées ou décrémentées en fonction de ce qu'on a spécifié dans **exprIncr** et le test est re-effectué, et ainsi de suite jusqu'à ce qu'il ait comme résultat **false** (ex : `for(let i=1; i<=10; i++) Afficher(i);`).

4. La boucle **for/of** : La syntaxe est : **for(exprInit of objetItérable) instruction;**
ou **for(exprInit of objetItérable) bloc.**

Elle s'utilise seulement avec les objets **itérables** (ceux contenant la fonction `Symbol.iterator()`) comme les tableaux, les sets, les chaînes de caractères, etc. A chaque itération, elle affecte la **valeur** d'une propriété à **exprInit** puis exécute l'instruction ou le bloc. Elle répète ce processus jusqu'à ce que toutes les propriétés de **objetItérable** aient été parcourues (ex :

```
for(const i of fruits){ //A supposer que fruits est un tableau
    console.log(i); //Affiche les valeurs respectives contenues dans le tableau
}
```

5. La boucle **for/in** : La syntaxe est : **for(exprInit in objet) instruction;**
ou **for(exprInit in objet) bloc.**

Elle s'utilise comme la boucle **for/of**, sauf qu'elle s'applique sur tout type d'objet, itérable ou pas. A chaque itération, elle affecte le **nom** d'une propriété à **exprInit** puis exécute l'instruction ou le bloc. (ex :

```
for(const i in fruits){ //A supposer que fruits est un tableau
    console.log(i); //Affiche 0, 1, 2, ... respectivement, les indexs du tableau
}
```

STRUCTURE DE CONTROLE : INSTRUCTIONS DE SAUT

Une instruction de saut ordonne à JavaScript de continuer l'exécution d'un script à un autre endroit. Utiliser certaines instructions de saut demande d'**étiqueter** les endroits où on veut effectuer ces sauts. Une **étiquette** est un nom (identifiant) qu'on attribue à une instruction ou à un bloc dans le but de le cibler par une instruction de saut. La syntaxe d'un étiquetage est : **identifiant : instruction;** ou **identifiant : bloc** (ex :

`truc : x = 2+5; //On a placé l'étiquette "truc" sur cette instruction`

`foo : { //on a donné un nom (foo) à ce bloc
 const pi = 3.14;`

- }). Il y a six types d'instructions de saut :

1. L'instruction **break** : La syntaxe est : **break;** ou **break étiquette;**.

Elle s'utilise dans une boucle **for**, **while** ou dans un **switch** et permet de quitter l'exécution de cette boucle ou structure de contrôle et revenir dans l'exécution globale du script ou à l'endroit où l'étiquette est déclarée. Cette étiquette doit impérativement dénoter la boucle qu'on veut quitter (ex :

```
for(let i = 0; i<10; i++){
    if(i==5) break; //sort de la boucle lorsque i vaut 5
    console.log(i);
}
boucleExt: for(let i = 0; i<5; i++){
    for(let j = 0; j<5; j++){
        if(i==2 && j==3) break boucleExt; //sort de la boucle 'boucleExt'
        console.log(`i = ${i}, j = ${j}`);
    }
}
```

2. L'instruction **continue** : La syntaxe est : **continue;** ou **continue étiquette;**.

Elle s'utilise dans une boucle **for** ou **while** et permet de redémarrer l'exécution de cette boucle. Dans une boucle **for**, elle incrémente la variable d'incrémentation avant de redémarrer la boucle

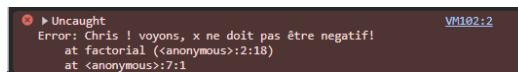
dès le début ou à partir de l'endroit où on a déclaré l'étiquette. Cette étiquette doit impérativement dénoter la boucle qu'on veut réitérer (ex :

```
for(let i = 0; i<10; i++){
    if((i%2)==0) continue; //itération suivante si i est pair
    console.log(i);
}
boucleExt: for(let i = 0; i<5; i++){
    for(let j = 0; j<5; j++){
        if(j==2) continue boucleExt; //itération suivante de 'boucleExt'
        console.log(`i = ${i}, j = ${j}`);
    }
}).
```

3. L'instruction `return` : La syntaxe est : `return expression;`. Elle s'utilise dans une fonction. Elle met fin à celle-ci et renvoie `expression` comme valeur de retour. Voir chapitre 7.
4. L'instruction `yield` : La syntaxe est : `yield expression;`. Voir chapitre 11.
5. L'instruction `throw` : On s'en sert pour explicitement lancer des exceptions, qui sont des erreurs pendant l'exécution (*runtime errors*). Une exception peut être gérée par un *gestionnaire d'exceptions* sans pour autant faire échouer l'exécution du script. S'il n'y a pas de gestionnaire d'exceptions dans le script, alors pas le choix, l'exécution du script s'interrompt et JavaScript le signale comme une erreur d'interprétation. La syntaxe est : `throw expression;`. Où `expression` s'agit de n'importe lequel des 12 types d'expressions (ex : `throw 5` empêche le script de s'exécuter s'il n'y a pas de gestionnaire d'exceptions et produit le message d'erreur  `Uncaught 5` dans la console). Généralement, on utilise un objet `Error` qui permet de présenter les erreurs de manière plus informative (ex :

```
function factorielle(x){
    if(x<0) throw new Error("Chris ! voyons, x ne doit pas être négatif!");
    let f;
    for(f = 1; x>1; f *= x, x--) ;
    return f;
}
factorielle(-4);
```

produit le message suivant dans la console si aucun gestionnaire d'exceptions n'est présent dans le script :



). On utilise souvent les lanceurs d'exceptions pour générer des erreurs pour certains comportements de nos scripts qui sont mauvais mais qui ne sont pas des erreurs du langage (ex : problèmes de connexion, erreurs de résolution de promesses, problèmes de sécurité, etc.).

6. L'instruction `try/catch/finally` : Généralement, lorsqu'un bout de code est susceptible de déclencher une exception, il est recommandé de l'utiliser dans un *gestionnaire d'exceptions* de sorte à ce qu'on soit sûr que l'erreur qui pourrait être lancée serait traitée par le gestionnaire adéquat et ne se propagerait pas jusqu'à produire une erreur d'interprétation. `try/catch/finally` s'agit de ce gestionnaire en JavaScript. La syntaxe est :

```
try{
    On place ici le code susceptible de lancer une exception. Si le code en lance une, cette exception est stocké dans un objet e puis passé en paramètre à catch()
}catch(e){
    On décide ici de quoi faire si une exception est lancée dans le try, on identifie ce message par l'objet e (e n'est qu'à titre d'exemple, on peut nommer cet objet comme on veut)
}finally{
    Le code qu'il y a ici s'exécute, que l'exception soit lancée ou pas. On peut omettre ce bloc car pas obligatoire
}
```

A chaque fois qu'on utilise un `throw` ou qu'on utilise une fonction qui en contient un, il est recommandé de l'utiliser dans un gestionnaire d'exceptions. Sinon, soit on risque de produire une erreur d'interprétation, soit l'erreur peut être accidentellement gérée par un `try/catch/finally` qui n'a pas été écrit pour elle et donc produire une confusion lors du débogage (ex :

```
function factorielle(x){ //cette fonction émet une exception
    if(x<0) throw new Error("Chris ! voyons, x ne doit pas être négatif!");
```

```

let f;
for(f = 1; x>1; f *= x, x--) ;
return f;
}

try{
  //Demande à l'utilisateur de rentrer un entier positif
  let n = Number(prompt("Entrez un entier positif svp", ""));
  //Calcule la factorielle
  let f = factorielle(n);
  //Affiche le résultat
  alert(n + " ! = " + f);
} //Si l'entier entré est négatif, l'exception émise est générée ici
catch(ex){
  alert(ex);
} //Au lieu d'afficher l'erreur dans la console, on l'affiche dans un popup).

try peut être utilisé seul, mais catch et finally doivent impérativement être utilisés avec un try.

```

INSTRUCTIONS DIVERSES

Les instructions diverses sont celles qui n'ont pas besoin d'être catégorisées. Il y en a 3 :

1. L'instruction **with** : Cette instruction permet de simplifier la syntaxe qu'il faut pour accéder aux propriétés d'un objet. La syntaxe est **with(objet){instructions}**. Par exemple, pour accéder à une des propriétés d'un objet **const car = {make: 'Toyota', model: 'Corolla', year: 2020};**, on écrit généralement **car.make**; . avec **with**, on peut simplement écrire :


```
with(car){
  make; //On accède directement à la propriété sans écrire « car. »
}. C'est comme un espace de noms en C++. Il est cependant déconseillé d'utiliser cette instruction.
```
2. L'instruction **debugger** : Elle est utilisée pour arrêter l'exécution d'un script à un point précis. Elle ouvre ensuite le débogueur et permet au développeur d'inspecter l'exécution du script jusqu'à ce point. La syntaxe est : **instruction debugger;** (ex : **let f = factorielle(n) debugger;**).
3. L'instruction **"use strict";** : Elle est utilisée pour activer le mode strict, dans l'ensemble du script si déclaré globalement, dans la portée du bloc si déclaré dans une fonction ou dans un quelconque bloc. La syntaxe est **"use strict";**. Pour savoir ce que c'est le mode strict, voir « *JavaScript: The definitive guide* ».

INSTRUCTIONS DE DECLARATION

Une déclaration permet de créer et de stocker des valeurs en mémoire vive. Il y a 4 types d'instructions de déclaration :

1. L'instruction de déclaration de variable : Voir chapitre 2.
2. L'instruction de déclaration de fonction : « **function nomFonction (paramètres) bloc** ». Voir chapitre 7 pour plus d'explications.
3. L'instruction de déclaration de classe : Voir chapitre 8.
4. L'instruction de déclaration de module : Voir chapitre 9.

CHAPITRE 5 : OBJETS

Un objet, c'est une liste de propriétés. La propriété d'un objet est une paire constituée de deux éléments, un **nom** (qui peut s'agir d'un identifiant, d'une chaîne de caractères ou d'un symbole) et une **valeur** (qui peut être de type primitif ou s'agir d'un autre objet) séparés par un « **:** ». Un objet est une instance de l'objet **Object**.

OBJETS : DECLARATION

Un objet peut être déclaré de trois manières :

1. Avec un littéral Object : Sa syntaxe est `{nom1: expr1, nom2: expr2, etc.}`. Où nom s'agit d'un identifiant, un string ou un symbole et expr s'agit de n'importe lequel des 12 types d'expressions (ex : `{nom: "christian", "solde du compte": 899364343};`). En écrivant ce littéral, on crée un objet qu'on peut ensuite stocker dans une variable si on veut le réutiliser (ex :

```
let livre = { //On a stocké cet objet dans la variable (et donc, l'objet) livre
    titre: "JavaScript",
    "sous-titre": "The Definitive Guide",
    auteur: {
        nom: "David Flanagan",
    }
};
//On stocke cet objet dans la constante (et donc, l'objet) a
const a = {id: 1, nbrVie: NaN, maFonction: function(x){return x**2;}};
```

Quand on stocke un objet dans une variable, on peut réassigner à cette variable une autre valeur.

Quand on utilise `const`, on ne peut pas (ex :

```
let a = {id: 12}; //a est une variable qui contient un objet
const b = {nbrVie: 3}; //b est une constante qui contient un objet
a = 3; //a ne contient plus un objet, mais une valeur de type Number
b = 5; //Erreur! On ne peut pas réaffecter une nouvelle valeur à un const).
```

On peut modifier la valeur d'une propriété préexistante d'un objet, peu importe s'il est stocké dans une variable ou dans une constante (ex :

```
a.id = 4; //valide
b.nbrVie = 9; //valide aussi.
```

Si un objet est stocké dans une variable, on peut lui rajouter d'autres propriétés sans problèmes, mais pas s'il est stocké dans une constante (ex : `livre.annee = 2020`; ajoute la propriété `annee` à l'objet `livre` mais `a.score = 2`; va générer une erreur).

On utilise cette notation pour :

- Créer un objet simple qui ne demande pas une logique complexe comme l'héritage, etc.
- Créer un objet qui contient des données corrélées (ex : une table de base de données).
- Créer un objet dont on connaît toutes les propriétés d'avance.
- Créer un objet à usage unique (en écrivant simplement le littéral sans le stocker).
- Créer un objet qui n'a pas besoin de prototypes.

2. Avec un constructeur : Un constructeur est une fonction dont le but est de créer un objet et initialiser toutes ses propriétés. Bien qu'il existe des constructeurs pour des objets natifs à JavaScript (ex : Date, Object, etc.), on peut créer ses propres constructeurs (voir chapitre 8). Pour créer un objet avec un constructeur, on utilise l'expression de création d'objet : `new Object(args)` (ex : `let obj = new Object();` //crée un objet sans propriétés : similaire à `{}`).

On utilise cette notation pour créer plusieurs **instances** d'un même objet. Ces instances ont les mêmes propriétés mais peuvent avoir des valeurs différentes.

3. Avec la fonction `Object.create()` : `create()` s'agit d'une des propriétés du prototype `Object`. Un prototype est un objet dont les propriétés sont héritées par d'autres. En JavaScript, tout objet hérite des propriétés d'un autre, ces autres héritent de celles d'autres et ainsi de suite. Ces objets qui partagent leurs propriétés avec l'objet qu'on compte créer forment une *chaîne de prototypes* à laquelle il pourrait accéder. La syntaxe est `Object.create(objet)`. La fonction prend en paramètre un objet. Il crée un objet vide et hérite à celui-ci les propriétés de l'objet qu'on lui a passé en argument (ex :

```
let obj1 = Object.create({x: 1, y: 2}); //obj1 hérite des propriétés x et y
let obj2 = Object.create(null); //obj2 n'hérite d'aucune propriété
let obj3 = Object.create(obj1); //obj3 hérite des propriétés de obj1.
```

On utilise cette notation pour :

- Créer des objets liés par prototypes (des objets qui peuvent accéder aux propriétés d'autres objets présents dans leur chaîne de prototypes).

- Cloner un objet (créer autre objet avec les mêmes propriétés et valeurs).
- Créer un objet sans prototype.

OBJETS : DEFINITION ET ACCES AUX PROPRIETES D'UN OBJET

En JavaScript, les propriétés d'un objet peuvent être évaluées de 2 manières après qu'ils aient été créés : leur valeur peut être lue ou modifiée. Lorsqu'on lit la valeur de la propriété d'un objet, on dit avoir accès à cette propriété. Lorsqu'on modifie la propriété et qu'on lui affecte une valeur pour la première fois, on dit qu'on définit cette propriété (on peut ensuite la redéfinir autant qu'on le souhaite).

Pour accéder aux propriétés d'un objet, on utilise l'expression d'accès à la propriété d'un objet. Lorsque le nom de la propriété à laquelle on veut accéder est un identifiant, on utilise la syntaxe : `objet.propriété` (ex : `obj3.x; //on lit la valeur de la propriété x contenue dans l'objet obj3`). Lorsque le nom de la propriété à laquelle on veut accéder est un string, on utilise la syntaxe : `objet["propriété"]` (ex : `obj2["pr x"];`).

En JavaScript, on peut accéder aux propriétés d'un objet *en chaîne*. Par exemple, soit l'objet suivant :

```
let o1 = {
  x: 1,
  o2: {
    y: "coucou",
    o3: {z: 3}
  }
};
```

L'objet o1 contient deux propriétés, x qui est un number et o2 qui est aussi un objet. L'objet o2 contient à son tour deux propriétés, y, un string et o3, un autre objet qui, lui, ne contient qu'une seule propriété z. Si on veut lire la propriété z, on doit écrire :

```
o1.o2.o3.z;
```

Pour définir une valeur à une propriété d'un objet, on utilise une expression d'assignation. L'opérande1 s'agit d'une expression d'accès à la propriété et l'opérande2 s'agit de n'importe lequel des 12 types d'expressions (ex :

```
obj3.x = 3; //on définit la propriété x à 3).
```

Accéder à une propriété qui n'existe pas ne lance pas une erreur. Il est évalué à `undefined`. Cependant, définir une propriété qui n'existe pas peut produire une erreur en fonction de si l'objet qui la contient est stockée dans une variable ou dans une constante.

OBJETS : SUPPRESSION DES PROPRIETES D'UN OBJET

Pour supprimer la propriété d'un objet, on utilise l'expression `delete`. Voir chapitre 3.

OBJETS : TESTS SUR LES PROPRIETES D'UN OBJET

On peut effectuer des tests sur la propriété d'un objet en le passant comme opérande dans les expressions logiques ou relationnelles. Voir chapitre 3.

OBJETS : PARCOURS DES PROPRIETES D'UN OBJET

On peut parcourir les propriétés d'un objet en utilisant la boucle `for/in` (voir chapitre 4) (ex :

```
let o = {x: 1, y: 2, "z": 3};
```

```
for(let p in o){ //parcours les propriétés de l'objet o
  console.log(p); //affiche "x", "y" et "z" respectivement, les noms des propriétés
}
```

, ou en utilisant l'une des 4 fonctions suivantes :

- `Object.keys(objet)` : Elle prend en paramètre un objet et retourne un tableau contenant les noms des propriétés itérables de cet objet.
- `Object.getOwnPropertyNames(objet)` : Elle prend en paramètre un objet et retourne un tableau contenant les noms des propriétés de cet objet à condition que ces noms soient des symboles.
- `Object.getOwnPropertySymbols(objet)` : Elle prend en paramètre un objet et retourne un tableau contenant les noms des propriétés itérables de cet objet
- `Reflect.ownKeys(objet)` : Elle prend en paramètre un objet et retourne un tableau contenant les noms des propriétés de cet objet, itérables ou pas, symboles ou pas.

Il est à noter que ce n'est pas tous les objets qui peuvent être parcourus. Par exemple, certains objets natifs de JavaScript comme `Object` ou `Date` ne peuvent pas être parcourus.

OBJETS : EXTENSION D'UN OBJET

Etendre un objet, c'est ajouter à ses propriétés, celles d'autres objets. On étend un objet en utilisant la fonction `Object.assign(objet1, objet, etc.)`. Elle prend en paramètre deux ou plusieurs objets ; `objet1` est l'objet qu'on veut étendre, le reste des objets sont ceux desquels on va copier les propriétés. Elle copie les propriétés de ces objets et les rajoute dans `objet1`.

On peut aussi étendre un objet avec les propriétés d'autres objets en utilisant une *expression d'extension ou spread* « `...objet` » (ex :

```
let position = {x: 0, y: 0};
let dimensions = {long: 100, larg: 75};
let rect = {...position, ...dimensions}; //x, y, Long et Larg sont rajoutés dans rect).
```

OBJETS : SERIALISATION D'UN OBJET

La sérialisation est un procédé permettant de transformer des données en un format de description (markup) qui facilite leur partage. Par exemple, le HTML est un format facile et très léger à envoyer sur internet. Grâce à lui, le navigateur peut reconstruire une page qui, si on devait l'envoyer tel quel, pèserait plusieurs centaines de mégaoctets. Les objets en JavaScript peuvent être sérialisés en un format spécial, appelé **JSON** (*JavaScript Object Notation*).

Pour sérialiser un objet, et donc, le traduire dans ce format, on utilise la fonction `JSON.stringify(objet)`. Cette fonction prend en paramètre un objet et retourne un string contenant toutes ses propriétés au format JSON (ex :

`JSON.stringify({x: 0, y: 0})` retourne la valeur '`{"x": 0, "y": 0}`'.

Enfin, pour désérialiser un objet préalablement sérialisé, on utilise la fonction `JSON.parse(string)`. Elle prend en paramètre un string et essaie de reconstruire l'objet qu'il retourne ensuite (ex :

`JSON.parse('{"x": 0, "y": 0}')` retourne l'objet `{x: 0, y: 0}`.

OBJETS : METHODES D'OBJETS

Tous les objets en JavaScript, excepté ceux qui sont explicitement créés sans prototypes, héritent des propriétés de leurs prototypes, et ces propriétés sont généralement des fonctions (ex : `objet.toString()`).

Pour avoir une liste complète des propriétés de l'objet `Object`, voir sur **MDN Web Docs**.

CHAPITRE 6 : TABLEAUX

Un tableau est une instance de l'objet `Array`. Il s'agit d'une liste de valeurs identifiées par un numéro (*index*) attribué automatiquement et implicitement. Les éléments d'un tableau peuvent être de types différents.

TABLEAUX : DECLARATION

Un tableau peut être déclaré de quatre manières au choix :

1. Avec un littéral Array : Sa syntaxe est `[expr1, expr2, etc.]`. Où `expr` s'agit de n'importe lequel des 12 types d'expressions (ex : `["christian", 899364343]`). En écrivant ce littéral, on crée un tableau de deux éléments qu'on peut ensuite stocker dans une variable si on veut le réutiliser (ex : `//un tableau qui contient un string, un number, un boolean, un objet et un tableau
let liste = ["hello", 1, true, {x: "joli", y: null}, [9, false]]`).

Comme pour les objets, quand on stocke un tableau dans une variable, on peut réassigner à cette variable une autre valeur. Quand on utilise `const`, on ne peut pas.

De même que pour les objets, on peut rajouter les éléments d'un tableau à un autre grâce à une expression d'extension (ex :

```
let a = [1, 2, 3];
let b = [...a, 4]; //b devient [1, 2, 3, 4]
let original = [1, 2, 3, 4];
let copie = [...original]; //On crée une copie du tableau « original ».
```

2. Avec un constructeur : Pour créer un objet avec un constructeur, on utilise l'expression de création d'objet : `new Array(taille)` ou `new Array(tableau)` (ex :
`let obj = new Array(10); //crée un tableau pouvant stocker 10 éléments`
`let obj1 = new Array(`A`); //crée un tableau contenant un élément : Un string`
`let obj2 = new Array(1, 2, 3, 4); //crée un tableau avec 4 éléments`). Lorsqu'on ne passe qu'un seul argument à ce constructeur, si cet argument est un `number`, il est considéré comme la taille du tableau. La taille du tableau est spécifiée à ce nombre, mais il ne contient encore aucun élément. S'il s'agit d'une valeur d'un autre type, il est rajouté comme un élément de ce tableau. Si l'on passe deux ou plusieurs arguments, ces arguments sont individuellement considérés comme des éléments de ce tableau. On utilise cette notation pour les mêmes raisons qu'en utilisant la première notation.
3. Avec la fonction `Array.of()` : Avec le constructeur `Array()`, on ne peut pas créer un tableau avec un seul élément de type `number`, car il serait par défaut considéré comme la taille du tableau. La méthode `Array.of()` permet justement de palier à ce problème et de créer un tableau avec `un` ou plusieurs éléments, qu'ils soient des `number` ou pas (ex :
`let x = Array.of(); //crée []; un tableau vide`
`let y = Array.of(10); //crée [10]; un tableau avec un seul argument numérique`
`let z = Array.of(1, 2, 3); //crée [1, 2, 3].`)
4. Avec la fonction `Array.from()` : Il se comporte comme une expression d'extension. Il prend un tableau en argument et retourne un autre tableau contenant ses éléments (ex :
`let copie = Array.from(original); //copie les éléments de ce tableau dans copie.`)

TABLEAUX : DEFINITION ET ACCES AUX ELEMENTS D'UN TABLEAU

Pour accéder aux éléments d'un tableau, on utilise l'expression d'accès à l'élément d'un tableau. On utilise la syntaxe : `nomTableau[index]`. Où l'`index` s'agit de n'importe quelle expression qui s'évalue en un `number` (ex :

```
tab[2]; //Lit le troisième élément du tableau tab
tab[2+5]; //S'évalue à tab[7] et le lit).
```

Les éléments d'un tableau sont indexés à partir de 0. Le premier élément est indexé à 0 ; le deuxième, 1 ; etc.

Pour définir une valeur à un élément d'un tableau, on utilise une expression d'assignation. L'opérande 1 s'agit d'une expression d'accès à l'élément d'un tableau et l'opérande 2 s'agit de n'importe lequel des 12 types d'expressions (ex : `tab[4] = 3; //on définit le cinquième élément à 3`).

Accéder à un élément qui n'existe pas ne lance pas une erreur. Il est évalué à `undefined`. Cependant, définir un élément qui n'existe pas peut produire une erreur en fonction de si le tableau qui le contient est stockée dans une variable ou dans une constante.

TABLEAUX : TABLEAUX DESORDONNES

Ils s'agissent de tableaux dont la taille n'est pas égale au nombre d'éléments (ex :

```
let a = new Array(5); //un tableau avec 0 éléments, mais on spécifie la taille à 5
a = []; //on réaffecte à la variable un tableau avec 0 éléments et de taille 0
a[1000] = 0; //on affecte la valeur 0 au 1001eme élément de a). Ils ne sont pas utiles mais il est bon d'avoir connaissance d'eux.
```

TABLEAUX : TAILLE

Pour lire la taille d'un tableau, on utilise une propriété qu'il hérite de son prototype : `length` (ex :

```
let a = [1,2,3,4,5]; //crée un tableau de 5 éléments
console.log(a.length); //affiche 5). Il faut aussi savoir qu'on peut modifier la taille d'un tableau en affectant une valeur à cette propriété (ex :
a.length = 3; //on réduit la taille de a, les deux derniers éléments sont supprimés).
```

TABLEAUX : AJOUT ET SUPPRESSION D'ELEMENTS D'UN TABLEAU

Pour rajouter un élément à un tableau,

- On peut juste affecter une nouvelle valeur à un nouvel index de ce tableau (ex :


```
let a = []; //on commence par un tableau vide
a[0] = "zero"; //puis on y ajoute des éléments. Le tableau devient ["zero"].
```
- On peut utiliser la propriété `push()`. Cette fonction rajoute les valeurs qu'on le passe en argument à la fin du tableau qui l'invoque (ex :


```
let a = [0, 3]; //on commence avec un tableau avec deux éléments
a.push(1, 2); //on rajoute deux éléments. Le tableau devient [0, 3, 1, 2].
```

Pour supprimer un élément d'un tableau,

- On peut utiliser une expression `delete` (ex : `delete a[0]`; //on supprime le 1^{er} elt de a).
- On peut réaffecter une nouvelle taille au tableau via `length`.
- On peut utiliser la propriété `pop()`. Cette fonction ne prend aucun argument, mais supprime le dernier élément du tableau qui l'invoque (ex :


```
let a = [0, 1, 2]; //on commence avec un tableau avec trois éléments
a.pop(); //supprime 2. Le tableau devient [0, 1]
a.pop(); //supprime 1. Le tableau devient [0].
```

TABLEAUX : PARCOURS DES ELEMENTS D'UN TABLEAU

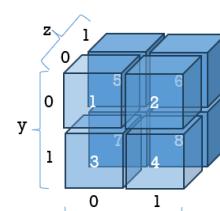
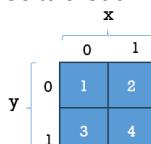
On peut parcourir les éléments d'un tableau en utilisant la boucle `for`, `for/in` ou `for/of`. Voir chapitre 4.

TABLEAUX : TABLEAUX MULTIDIMENSIONNELS

En JavaScript, un tableau multidimensionnel est un tableau dont les éléments sont d'autres tableaux (ex :

```
let a = [
    [1, 2], //Ligne y0 avec deux colonnes x0 et x1
    [3, 4] //Ligne y1 avec deux colonnes x0 et x1
];
//il s'agit d'une matrice 2D de deux colonnes et deux lignes

let b = [
    //Profondeur z0 (La matrice de devant)
    [1, 2], //Première Ligne : z0y0
    [3, 4] //Deuxième Ligne : z0y1
],
    //Profondeur z1 (La matrice de derrière)
    [5, 6], // Première Ligne : z1y0
    [7, 8] // Deuxième Ligne : z1y1
];
//il s'agit d'une matrice 3D de 2 colonnes, 2 lignes et une profondeur de 2).
```



Pour utiliser un élément de ce tableau, on peut par exemple écrire :

```
a[0][0]; //on lit 1
a[0][1]; //on lit 2. La syntaxe d'accès à chaque élément est : a[y][x]
a[1][0]; //on lit 3
a[1][1]; //on lit 4
```

```
b[0][0][0]; //on lit 1
b[0][0][1]; //on lit 2. La syntaxe d'accès à chaque élément est : b[z][y][x]
b[0][1][0]; //on lit 3
b[0][1][1]; //on lit 4
b[1][0][0]; //on lit 5
b[1][0][1]; //on lit 6
b[1][1][0]; //on lit 7
b[1][1][1]; //on lit 8
```

Pour parcourir un tableau multidimensionnel, on utilise plusieurs boucles **for**, **for/in** ou **for/of** en fonction de comment il est composé. Si c'est un tableau 2D, on en utilise deux, si c'est un tableau 3D, on en utilise trois, ainsi de suite (ex :

Pour le tableau 2D :

```
for(let y of a){ //pour chaque ligne y,
  for(let x of y){ //parcourir tous les éléments x
    console.log(x);
  }
}
```

Pour le tableau 3D :

```
for(let z of b){ //pour chaque profondeur z du tableau b,
  for(let y of z){ //parcourir chaque ligne y,
    for(let x of y){ //et pour chaque ligne y, parcourir chaque élément x
      console.log(x);
    }
}
```

). Généralement, quand on programme dans la vraie vie, on ne va pas au-delà des tableaux 3D.

TABLEAUX : METHODES DE TABLEAUX

Il existe plusieurs méthodes qu'on peut appeler sur des tableaux pour les manipuler. Pour avoir une liste complète des propriétés de l'objet [Array](#), voir sur [MDN Web Docs](#).

CHAPITRE 7 : FONCTIONS

Une fonction est une instance de l'objet [Function](#). Une fonction est un bloc d'instructions qu'on définit une fois, mais qu'on peut réutiliser plusieurs fois en l'invoquant.

FONCTIONS : DEFINITION

On définit une fonction en utilisant, en fonction du besoin, une expression de définition de fonction ou une instruction de déclaration de fonction :

1. `function nomFonction (paramètres) bloc` ou
2. `function (paramètres) bloc` ou
3. `(paramètres) => instructions`

La première notation s'agit d'une **instruction de déclaration de fonction**. Elle est utilisée pour déclarer des fonctions de manière classique (des routines), comme dans d'autres langages (ex :

```
function distance(x1, y1, x2, y2){
  return Math.sqrt((x2-x1)**2 + (y2-y1)**2);
```

). En JavaScript, toutes les fonctions déclarées avec cette notation sont hoistables. Elles peuvent être utilisées avant d'être déclarées à condition que la déclaration et l'utilisation soient dans la même portée.

La deuxième notation s'agit d'une **expression de définition de fonction**. Une expression de définition de fonction est évaluée comme une valeur, c'est-à-dire, elle a pour but d'être utilisée comme opérande dans une expression. Elle est souvent stockée dans une variable ou liée à une propriété d'un objet (dans ce cas, on l'appelle « **méthode** »), mais elle peut être passée en argument à une autre fonction, être utilisée dans une expression d'accès à la propriété d'un objet, etc. (ex :

```
//une expression de définition de fonction qui est stockée dans une constante
const carré = function(x){
    return x**2;
};

//on peut lui donner un nom s'il doit faire appel à elle-même dans son corps
const f = function fact(x){
    if (x<=1) return 1;
    else return x*fact(x-1);
};

//elle peut être passée comme argument à une autre fonction
[3,2,1].sort(function(a,b){
    return a-b;
});
```

). Une fonction déclarée avec cette notation n'est pas hoistable, Lorsqu'elle est stockée dans une variable, elle n'est utilisable que dans la portée de cette variable. Lorsqu'elle est passée en argument à une autre fonction, elle est immédiatement exécutée et si en plus, elle retourne une valeur, celle-ci est passée en argument. Il est recommandé de stocker ce genre d'expressions dans une constante.

La troisième notation est un sucre syntaxique à l'expression de définition de fonction. On l'appelle aussi : « **notation fléchée** » (ex :

```
//forme standard d'une fonction fléchée
const somme = (x, y) => {
    return x+y;
```

). Si le corps d'une fonction fléchée ne consiste qu'en une instruction `return`, on peut omettre ce mot-clé (ex : `const somme = (x, y) => x+y;`). Si la fonction fléchée n'a qu'un seul paramètre, on peut omettre les parenthèses (ex : `const polynome = x => x*x + 2*x + 3;`). Si par contre, la fonction fléchée n'a aucun paramètre, il faut quand même mettre des parenthèses vides (ex : `const a = () => 1;`).

En JavaScript, une fonction peut être contenue dans une autre. Lorsqu'une fonction B est contenue à l'intérieur d'une autre fonction A, B peut accéder aux paramètres de A ainsi que toutes les variables qui y sont déclarées (ex :

```
function hypotenuse(a, b){
    function carre(x) {return x*x;}
    return Math.sqrt(carre(a) + carre(b));
}.
```

Lorsqu'une fonction est déclarée comme propriété d'un objet, on peut utiliser à l'intérieur de celui-ci, le mot-clé `this` pour faire référence à l'objet qui le contient (ex :

```
let o = { //un objet o.
    m: function(){ //méthode m définit dans l'objet
        let self = this; //self contient une référence à l'objet o
        this === o; //s'évalue à true. this denote l'objet o
        function f(){ //cette fonction est contenue dans m
            this === o; //false: "this" ne fonctionne pas dans les fonctions imbriquées, mais
            self === o; //true: à travers self, on peut l'utiliser dans une fonction imbriquée
        }
    }
};
```

FONCTIONS : INVOCATION

Invoquer une fonction, c'est ordonner à JavaScript de l'exécuter à l'endroit souhaité. On peut invoquer une fonction, en JavaScript, de 4 manières :

1. Avec une expression d'appel de fonction : On utilise cette expression pour appeler une fonction déclarée avec une instruction de déclaration de fonction ou une expression de définition de fonction stockée dans une variable. On écrit `nomFonction(args)` ou `nomFonction.?args` (ex : `hypotenuse(3,2); //appelle la fonction, renvoie une erreur si elle n'existe pas` `hypotenuse.?6,4); //appelle la fonction, renvoie undefined si elle n'existe pas`).
2. Avec une expression d'appel de méthode : On utilise cette expression pour appeler une méthode, une fonction déclarée comme propriété d'un objet. On écrit `nomObjet.nomMéthode(args)` ou `nomObjet.?nomMéthode(args)` (ex : `console.log(`hey !`); //appelle la méthode, renvoie une erreur si elle n'existe pas` `console.?clear(); //appelle la méthode, renvoie undefined si elle n'existe pas`).

On peut invoquer les méthodes d'un objet en chaîne. Par exemple, soit l'objet suivant :

```
const obj = {
  A: function(){
    return {
      B: function(){
        return {C: 4};
      }
    };
  }
};
```

On voit que l'objet `obj` contient une propriété `A` qui est méthode. Cette méthode retourne un objet qui à son tour contient une autre méthode `B` qui, lui, retourne un objet qui contient la propriété `C`, un `number`. Sachant cela, on peut accéder à `C` en invoquant les deux fonctions en chaîne. On écrit `obj.A().B().C;`. La méthode `A()` est exécutée en premier. Sa valeur de retour permet ensuite d'appeler `B()` et enfin, la valeur de retour de celle-ci permet d'accéder à `C`.

On peut définir une méthode et l'appeler immédiatement (ex :

`let dixAuCarré = (function(x){return x*x;}(10));`. Dans ce cas, on utilise uniquement cette notation. L'expression `(10)` qu'on a rajoutée à la fin correspond aux paramètres qu'on lui passe.

3. Avec un constructeur : Un constructeur est une fonction dont le but est de créer un objet et initialiser toutes ses propriétés. On invoque un constructeur pour ce seul but en utilisant une expression de création d'objet (ex : `let ID = new Object({prenom: "heïdi", postnom: "nade"});`).
4. Avec les méthodes `call()` et `apply()` : Les fonctions en JavaScript sont des objets, et comme tout objet, ils héritent des méthodes de leurs prototypes. L'un des prototypes des fonctions, l'objet `Function`, définit ces 2 méthodes qu'on peut appeler pour invoquer une fonction. Pour avoir une liste complète des propriétés de l'objet `Function`, voir sur [MDN Web Docs](#).

FONCTIONS : ARGUMENTS ET PARAMETRES

JavaScript n'exige pas de spécifier un type pour les paramètres d'une fonction. Il ne vérifie même pas le nombre d'arguments qu'on a passé à une fonction lors de son invocation. Lorsqu'une fonction est appelée avec moins de paramètres qu'il a été déclaré, les paramètres ignorés sont soit initialisés à leur valeur par défaut, soit initialisés à `undefined` (ex :

```
Function maFonction(o, a){
  return a+o;
}

let o = 1;
let a = maFonction(o); //va fonctionner.
```

On peut utiliser aussi une expression d'extension `(...)` dans la déclaration d'une fonction pour déclarer une fonction qui peut être invoquée avec plus d'arguments qu'il a été défini (ex :

```
/*Cette fonction prend un ou plusieurs arguments et retourne Le plus grand*/
function max(premiereValeur=-Infinity, ...reste){
```

```

let maxValeur = premiereValeur;
//on itère à travers le reste des paramètres pour trouver le plus grand
for(let n of reste){
  if (n > maxValeur){
    maxValeur = n;
  }
}
//retourne la plus grande valeur
return maxValeur;
}

max(1, 10, 100, 2, 3, 1000, 4, 5, 6); //retourne 1000). Dans cet exemple, la variable utilisée
dans l'expression d'extension est un tableau qui contient tous les autres arguments en trop qu'on a passé à
la fonction lors de son invocation.

```

On peut utiliser aussi une expression d'extension (...) lors de l'invocation d'une fonction pour passer tous les éléments d'un tableau ou les propriétés d'un objet comme paramètres individuels à une fonction (ex :

```

let numbers = [5, 2, 10, -1, 9, 100, 1];
Math.min(...numbers); //retourne -1).

```

FONCTIONS : METHODES DE FONCTIONS

Il existe plusieurs méthodes qu'on peut appeler sur les fonctions. Pour une liste complète des propriétés de l'objet **Function**, voir sur [MDN Web Docs](#).

CHAPITRE 8 : CLASSES

Lorsqu'on veut créer plusieurs instances d'un même objet, c'est-à-dire, des objets qui ont les mêmes propriétés mais qui peuvent avoir des valeurs différentes, on a vu qu'on pouvait utiliser un constructeur. Il s'agit d'une fonction dont le but est de créer un objet et initialiser toutes ses propriétés. Bien qu'il existe des constructeurs pour des objets natifs à JavaScript (ex : Date, Object, etc.), on a vu qu'on pouvait créer ses propres constructeurs. Lorsqu'on crée son propre constructeur, on implémente une classe.

Un constructeur est une fonction qui retourne un objet (autrement appelé fonction *factory*), mais toute fonction qui retourne un objet n'est pas un constructeur. En effet, un constructeur est une fonction qui se doit d'être invoquée via une expression de création d'objet (`new`). Pour cette invocation aboutisse, elle doit être implémentée de la manière suivante :

1. On crée un objet prototype,
2. On rajoute à ce prototype, les propriétés spécifiques au constructeur qu'on veut créer,
3. On retourne l'objet prototype créé contenant, en plus les propriétés que l'on a rajouté.

(ex :

```

function Identite(nom, postnom){
  //On crée un objet prototype de l'objet Object
  let prototype = Object.create();

  //on rajoute deux propriétés qu'on veut propre à ce constructeur
  prototype.nom = nom;
  prototype.postnom = postnom;

  //on peut rajouter une méthode si on veut
  prototype.afficher = () => nom+' '+postnom;

  //on retourne l'objet, étendu
  return prototype;
}
//on peut maintenant créer un objet en utilisant notre constructeur (classe) Identite
let monIdentite = new Identite('koyo', 'nsungu');
monIdentite.afficher(); //retourne : koyo nsungu).

```

Vu qu'une classe est une fonction, et qu'une fonction est un objet, et que tout objet a une propriété nommée `prototype`, on peut réécrire cette classe de la manière suivante :

```
function Identite(nom, postnom){
    this.nom = nom; //on met ici les propriétés qui seront initialisés par le constructeur
    this.postnom = postnom;
}
//on affecte à prototype un objet contenant les autres propriétés de notre classe
Identite.prototype = {
    afficher : () => this.nom + ' ' + this.postnom,
};

//on peut maintenant créer un objet en utilisant notre constructeur (classe) Identite
let monIdentite = new Identite('koyo', 'nsungu');
monIdentite.afficher(); //retourne : koyo nsungu
```

Cependant, cette manière de faire n'est pas naturelle. Pour savoir que telle ou telle fonction est en fait une classe, il faut que le programmeur prenne le temps d'analyser en long et en large la fonction. Pour pallier à ce problème, on a créé une syntaxe plus naturelle, utilisant le mot-clé « `class` ».

CLASSES : DECLARATION AVEC LE MOT-CLE « CLASS »

Lorsqu'on utilise le mot-clé « `class` », la syntaxe commence à ressembler de plus en plus à celle d'autres langages. Une classe est constituée d'une fonction `constructor()` qui initialise les propriétés qui doivent l'être lors de l'instanciation avec `new`, ainsi que d'autres propriétés qu'on lui rajoute (ex :

```
class Identite {
    constructor(nom, postnom){ //notez qu'on a pas écrit function constructor()
        this.nom = nom;
        this.postnom = postnom;
    }
    afficher = () => nom + ' ' + postnom; //notez qu'on a pas écrit let, const ou var
}
```

`let monIdentite = new Identite('koyo', 'nsungu');` `monIdentite.afficher(); //retourne : koyo nsungu`. Toutes les propriétés qu'on a créées dans la fonction `constructor()` sont automatiquement rajoutées à la classe identité. Cette notation est beaucoup plus naturelle et nous permet de faire de l'orienté-objet comme dans d'autres langages. Cependant, il est à noter qu'une classe n'est dans tous les cas qu'une fonction, elle hérite donc directement des propriétés de l'objet `Function`, `Object` et celles de tous les objets qui se trouvent dans sa chaîne de prototypes.

On peut créer des **méthodes statiques**, c'est-à-dire des fonctions qu'on ne peut invoquer qu'à partir du constructeur d'une classe, en utilisant le mot-clé `static`. Ces méthodes sont attachées à la classe elle-même, plutôt qu'à ses instances, ce qui signifie qu'elles peuvent être appelées sans avoir besoin de créer un objet de cette classe. Par exemple :

```
class Identite {
    constructor(nom, postnom){
        this.nom = nom;
        this.postnom = postnom;
    }
    static afficherInfo(){ //on ne peut pas utiliser this dans une méthode statique
        return 'Ceci est une méthode statique.';
    }
}
```

`console.log(Identite.afficherInfo()); // Ceci est une méthode statique.`. On s'en sert pour créer des fonctions utilitaires (utility), des fonctions factory, ou des fonctions singletons. Pour en savoir plus sur ces termes, voir le livre « *Pro JavaScript Design Patterns* ».

En plus des méthodes statiques, on peut aussi définir des **getters** et des **setters** pour contrôler l'accès aux propriétés d'une classe. Les getters, utilisant le mot-clé « **get** », permettent de récupérer la valeur d'une propriété privée ou protégée, tandis que les setters, utilisant le mot-clé « **set** », permettent de la modifier tout en offrant un contrôle supplémentaire (ex :

```
class Identite {
    constructor(_nom, _postnom){
        this.nom = _nom; //initialisation des champs (propriétés)
        this.postnom = _postnom;
    }
    get Nom(){
        return this.nom;
    }
    set Nom(valeur){
        if (valeur.length > 0){
            this.nom = valeur;
        } else {
            console.log("Le nom ne peut pas être vide.");
        }
    }
    get Postnom(){
        return this.postnom;
    }
    set Postnom(valeur){
        this.postnom = valeur;
    }
}

let monIdentite = new Identite("Charbi", "Kabandanyi");
console.log(monIdentite.Nom); //accès avec le getter. Affiche : charbi
monIdentite.Nom = "NouveauNom"; //modification avec le setter.
console.log(monIdentite.nom); // accès sans le getter. Affiche : NouveauNom
monIdentite.nom = ""; //modification sans le setter.). Quand on utilise le mot-clé set ou get, on peut appeler le setter ou le getter comme s'il s'agissait d'une propriété (ex : on a écrit monIdentite.Nom au lieu de monIdentite.Nom()).
```

En ce qui concerne les **champs publics, privés et statiques**, voici un aperçu :

1. **Champs publics** : Il s'agit des propriétés ou des méthodes directement accessibles et modifiables à l'extérieur de la classe. Par défaut, toutes les propriétés d'une classe en JavaScript sont publiques.
2. **Champs privés** : Ces propriétés sont destinées à être utilisées uniquement à l'intérieur de la classe. En JavaScript, on peut les créer en les préfixant par un symbole `#`, une fonctionnalité introduite avec ES2022 (ex :

```
class Identite {
    #nom; //création de champs privés
    #postnom;
    constructor(nom, postnom){
        this.#nom = nom;
        this.#postnom = postnom;
    }
    getNom(){
        return this.#nom;
    }
    setNom(nouveauNom){
        this.#nom = nouveauNom;
    }
}

let identite = new Identite("Kevin", "Musungu");
console.log(identite.getNom()); //accès avec le getter. Affiche « Kevin »
identite.setNom("NouveauNom"); //modification avec le setter
```

```
console.log(identite.getNom()); //accès avec le getter. Affiche « NouveauNom »
console.log(identite.#nom); //Erreur!, on ne peut pas lire un champ privé en direct).
Vous voyez qu'ici, on a pas utilisé le mot-clé set ou get, d'où on doit appeler identite.getNom()
par exemple et non pas identite.getNom.
```

3. **Champs statiques** : Similaires aux méthodes statiques, les champs statiques sont des propriétés associées à la classe elle-même et non aux instances. On les déclare en utilisant le mot-clé **static** (ex :

```
class Identite {
    static pays = "Congo"; //champ statique
    constructor(nom, postnom){
        this.nom = nom;
        this.postnom = postnom;
    }
    static obtenirPays(){
        return Identite.pays;
    }
}

console.log(Identite.pays); //affiche : Congo
console.log(Identite.obtenirPays()); //affiche : Congo).
```

CLASSES : AJOUTER DES METHODES A UNE CLASSE EXISTANTE

On peut rajouter des méthodes et des propriétés à une classe, même après l'avoir déclaré (quoi qu'il ne soit pas recommandé de le faire, sauf si on utilise la notation avec les fonctions). Il suffit de les rajouter à l'objet **prototype** (ex : `Identite.prototype.AfficherNom = () => this.nom; //rajoute une méthode`).

CLASSES : SOUS-CLASSES

Une **sous-classe** est une classe qui hérite des propriétés et des méthodes d'une autre classe, appelée la classe parente ou *super-classe*. En JavaScript, il existe deux manières de créer des sous-classes, notamment en utilisant la notation des fonctions ou les mots-clés modernes **extends** et **super** :

1. **Création de sous-classes avec la notation des fonctions** : Avant l'introduction de la syntaxe **class**, JavaScript reposait sur l'utilisation des fonctions constructeurs et des prototypes pour créer des sous-classes (ex :

```
function Personne(nom, age){ //création de la classe
    this.nom = nom;
    this.age = age;
}

Person.prototype.afficher = function(){ //rajout des méthodes
    console.log(`Nom: ${this.nom}, Age: ${this.age}`);
};

//Création de la sous-classe
function Etudiant(nom, age, classe){ //Appel du constructeur de la classe parente
    Personne.call(this, nom, age); //tu te souviens de cette méthode, non ? 😊
    this.classe = classe;
}

//Héritage du prototype de la classe parente
Etudiant.prototype = Object.create(Person.prototype);

//Définition d'une méthode propre à Étudiant
Etudiant.prototype.afficherClasse = function(){
    console.log(`Classe: ${this.classe}`);
};
```

```
let etudiant = new Etudiant("Jean", 20, "Mathématiques");
etudiant.afficher(); //Nom: Jean, Age: 20
etudiant.afficherClasse(); //Classe: Mathématiques.
```

Ici, `Etudiant()` est la sous-classe qui hérite de `Personne()`. Nous avons utilisé `Personne.call()` pour exécuter le constructeur parent dans le contexte de `Etudiant()`, et `Object.create()` pour hériter du prototype de `Personne()`.

2. **Création de sous-classes avec les mots-clés `extends` et `super`**: La syntaxe ES6 introduit une approche plus simple et plus naturelle pour créer des sous-classes en utilisant les mots-clés `extends` et `super`. Cela permet de faciliter l'héritage entre classes tout en rendant le code plus lisible (ex

```
class Personne {
  constructor(nom, age){
    this.nom = nom;
    this.age = age;
  }
  afficher(){
    console.log(`Nom: ${this.nom}, Age: ${this.age}`);
  }
}

class Etudiant extends Personne {
  constructor(nom, age, classe){
    super(nom, age); //appelle le constructeur de Personne (classe parente)
    this.classe = classe;
  }
  afficherClasse(){
    console.log(`Classe: ${this.classe}`);
  }
}
```

```
let etudiant = new Etudiant("Marie", 22, "Physique");
etudiant.afficher(); //Nom: Marie, Age: 22
etudiant.afficherClasse(); //Classe: Physique).
```

Le mot-clé `extends` est utilisé pour indiquer que `Etudiant` est une sous-classe de `Personne`. Le mot-clé `super` est nécessaire pour appeler le constructeur de la classe parente à l'intérieur du constructeur de la sous-classe.

Il existe un autre concept parallèle aux sous-classes : La **délégation**. C'est le mécanisme par lequel une méthode ou une propriété non trouvée sur un objet spécifique est automatiquement recherchée dans le prototype de cet objet, et ainsi de suite dans la chaîne des prototypes jusqu'à ce que la méthode ou la propriété soit trouvée, ou que la chaîne atteigne `null`. La recherche de cette méthode ou propriété est déléguée de prototype en prototype jusqu'à ce qu'il elle est trouvée (ex :

```
let parent = {
  saluer(){
    console.log("Bonjour de la part du parent !");
  }
};
```

```
//délégation : création d'un objet contenant les propriétés d'un autre objet
let enfant = Object.create(parent);
enfant.saluer(); //Bonjour de la part du parent !).
```

Dans cet exemple, l'objet `enfant` n'a pas de méthode `saluer()`, donc JavaScript délègue la recherche au prototype, qui est `parent`. Vu que celui-ci en a, elle est exécutée.

CHAPITRE 9 : MODULES

Les modules sont une fonctionnalité clé de JavaScript moderne qui permet de mieux organiser et structurer le code en le divisant en fichiers distincts, appelés **modules**. Chaque module contient du code spécifique

à une fonctionnalité ou à une partie de l'application et peut exporter et importer des parties de code entre différents fichiers. Cela améliore la réutilisabilité et la maintenabilité du code, en évitant des conflits entre les variables ou fonctions dans l'ensemble du projet.

IMPORTATION ET EXPORTATION DE MODULES

Avec l'introduction des modules ES6, JavaScript permet d'utiliser les mots-clés `export` et `import` pour gérer les modules. Voici comment cela fonctionne :

1. **Exportation d'un module** : Le mot-clé `export` permet de rendre une fonction, une classe, une variable ou une constante accessible depuis un autre fichier. Il existe deux types d'exportations : *l'exportation nommée* et *l'exportation par défaut* (ex :

Exportation nommée (Soit un fichier `math.js`)

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

`export const PI = 3.14159;`. Dans cet exemple, la fonction `addition()` et la constante `PI` sont exportées, et peuvent être importées et utilisées dans un autre fichier.

On peut aussi déclarer nos fonctions normalement, puis les exporter une fois pour toute :

```
function addition(a, b){
    return a+b;
}
const PI = 3.14159;
export{addition, PI}; //on importe deux modules en une fois.
```

Exportation par défaut (Soit un fichier `division.js`)

```
export default function (a, b){
    return a/b;
}
```

. Avec `export default`, vous pouvez exporter un élément principal par module (une fonction, une variable, etc.), ce qui est utile lorsqu'un module est destiné à n'avoir qu'une seule fonctionnalité principale).

2. **Importation d'un module** : Pour utiliser du code d'un autre module, le mot-clé `import` est utilisé.

Importation d'une exportation nommée :

```
import {addition, PI} from './math.js';
console.log(addition(2, 3)); //affiche 5
console.log(PI); // 3.14159. L'importation nommée nécessite d'indiquer exactement le nom des variables ou des fonctions que l'on veut importer dans les accolades.
```

On peut aussi importer tous les éléments de ce module en écrivant :

```
import * from './math.js';
```

On peut aussi importer des éléments de ce module et les stocker dans un objet en utilisant le mot-clé `as` : `import * as monObjet from './math.js';`

On peut aussi importer ce module, même s'il n'a pas de mot-clé `export` dans ses déclarations. Dans ce cas, On écrit : `import './math.js';`.

On peut importer et renommer un élément de ce module en écrivant par exemple :

```
import {PI as pi} from './math.js';
```

Importation d'une exportation par défaut :

```
import diviser from './division.js';
console.log(division(10, 2)); //affiche 5. Dans le cas de l'exportation par défaut, il n'est pas nécessaire d'utiliser des accolades, et le nom peut même être différent (ex : diviser) de celui utilisé dans le fichier d'origine.
```

MODULES NATIFS ET MODULES EXTERNES

JavaScript prend en charge les **modules natifs** à partir de l'ES6. Ces modules fonctionnent directement dans les navigateurs modernes ou dans un environnement comme Node.js, qui prend également en charge un autre système de modules, appelé **CommonJS** (Il s'agit d'une norme dictant comment développer des programmes JavaScript pouvant être exécutés en dehors du navigateur Web, comme des serveurs Web).

1. **Modules natifs ES6** : Ils utilisent `import` et `export` comme expliqué plus haut. Principalement utilisés dans les applications web modernes.

2. **Modules CommonJS (Node.js)** : Ils utilisent `module.exports` et `require()` (ex :

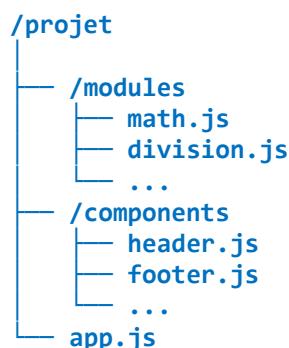
```
//En Node.js
//dans le fichier math.js
module.exports = {
  addition(a, b){
    return a + b;
  },
  PI: 3.14159
};

//dans le fichier app.js
const {addition, PI} = require('./math.js'); //on verra ça dans node.js
console.log(addition(2, 3)); // 5
console.log(PI); //3.14159). Dans Node.js, l'utilisation de la propriété module.exports et de la fonction require() est standard, bien que l'ES6 avec import et export soit aussi supporté avec certaines configurations.
```

UTILISATION DE MODULES POUR MIEUX ORGANISER LE CODE

L'utilisation de modules permet d'organiser les projets de grande taille en divisant le code en composants plus petits, chacun étant responsable d'une fonctionnalité distincte. Cela favorise une architecture modulaire et claire, dans laquelle chaque module peut être testé et maintenu indépendamment.

Un exemple d'organisation de projet avec des modules pourrait ressembler à ceci :



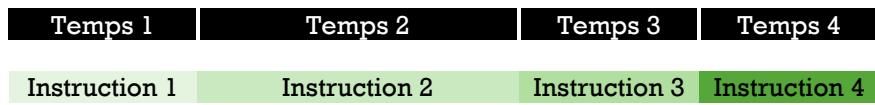
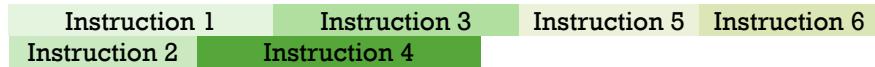
Ici, le répertoire `/modules` contient les fonctionnalités mathématiques exportées sous forme de modules, tandis que `/components` contient des modules spécifiques à l'interface utilisateur.

CHAPITRE 10 : GENERATEURS ET ITERATEURS

Très peu couramment utilisés, on n'en parle pas ici. Voir le livre « *JavaScript: The definitive guide* ».

CHAPITRE 11 : PROGRAMMATION ASYNCHRONE

Les programmes informatiques s'exécutent généralement d'une des deux manières existantes : de manière **synchrone** et **asynchrone**. Un programme s'exécute de manière synchrone lorsqu'il est lu une instruction à la fois. Il s'exécute de manière asynchrone lorsqu'il peut être lu plusieurs instructions en une fois.

Exécution synchrone**Exécution asynchrone**

Dans l'illustration ci-haut, on voit que lorsqu'un programme est exécuté de manière synchrone, à chaque temps, correspond l'exécution d'une et une seule instruction. Par défaut, tous les langages de programmation suivent ce modèle.

Cependant, on voit que lorsqu'un programme est exécuté de manière asynchrone, il y a quelques différences :

- Premièrement, le programme suit plusieurs fils d'exécution (ou threads), dans le cadre de cette illustration, il y a deux threads.
- Aussi, les instructions s'exécutent en parallèle sur les deux threads, il y a des fois où deux instructions sont exécutées en un seul temps

Cette manière de faire a plusieurs avantages (ex : un même programme peut lire un fichier et écrire un autre en même temps) mais aussi des désavantages (lorsque deux instructions manipulent une même donnée. Il peut y avoir risque de course de données (data race) qui peut mener à un blocage du programme).

Heureusement, JavaScript est non bloquant. Il s'exécute sur un seul thread et simule l'asynchronisme de la manière suivante : lorsqu'un script javascript est exécuté, toutes les parties qui doivent être exécutées de manière asynchrone sont ignorées. C'est lorsque l'exécution du script est terminée que commence l'exécution de ces instructions qui étaient ignorées. En réalité, cela se passe tellement vite (quelques millisecondes) qu'on a l'impression qu'elles sont exécutées en même temps que le script principal.

1 : Phase synchrone

Toutes les instructions asynchrones sont ignorées

2 : Phase asynchrone

Toutes les instructions asynchrones ignorées lors de la première phase sont exécutées dans l'ordre du premier venu, premier servi

La programmation asynchrone permet à JavaScript de gérer des tâches qui prennent du temps sans bloquer le flux d'exécution principal. Cela est essentiel pour des opérations comme l'accès à des API, la lecture de fichiers, ou des requêtes réseau.

Historiquement, la programmation asynchrone en JavaScript a commencé avec l'utilisation des **fonctions callbacks**. Ensuite, les **Promesses** sont apparues, offrant un meilleur contrôle et une meilleure lisibilité du code. Enfin, **async/await** a rendu l'asynchronisme encore plus simple et plus proche du style synchrone.

PROGRAMMATION ASYNCHRONE AVEC DES CALLBACKS

Un **callback** est une fonction A qui est passée en paramètre à une autre fonction B et qui est appelée au sein de celle-ci lorsque la tâche asynchrone est terminée. En termes simples, on dit à la fonction B, exécute-toi, mais à un certain moment de ton exécution, exécute aussi la fonction A que je t'ai donné. Les callbacks étaient la première méthode pour gérer l'asynchronisme en JavaScript (ex :

```
function chargerDonnées(url, callback){ //cette fonction est appelée callbackHandler
    setTimeout(() => { //cette fonction est native à JavaScript
        const données = `Données chargées depuis ${url}`;
        callback(données);
    }, 2000);
}
chargerDonnées('https://api.exemple.com', (données) => {
    console.log(données); //affiche les données après 2 secondes
});
```

Lorsqu'on invoque cette fonction, elle est ignorée et exécutée à la phase asynchrone

Cependant, l'utilisation de callbacks peut vite devenir compliquée et mener à ce qu'on appelle le *callback hell* : une situation où plusieurs opérations asynchrones imbriquées conduisent à un code difficile à lire et à maintenir (ex :

```
fonction1(() => {
  fonction2(() => {
    fonction3(() => {
      fonction4(() => {
        // et ainsi de suite...
      });
    });
  });
});
```

`fonction1()` appelle un callback `fonction2()` qui elle-même appelle un autre, `fonction3()` qui en appelle un autre `fonction4()` et ainsi de suite. Le code devient difficile à comprendre, surtout lorsqu'il y a des erreurs à gérer à chaque niveau.

Faire de l'asynchronisme avec des callbacks, c'est comme avoir la discussion suivante avec une fonction :

- Moi : "Hey ! fonction A, peux-tu t'exécuter en tâche de fond ?"
- A : "Bien sûr ! et je fais quoi après que j'ai fini de m'exécuter ?"
- Moi : "Tiens cette fonction (callback), lorsque tu as fini de t'exécuter, invoque-là et je saurais."
- A : "OK, mais sache qu'en cas d'erreur, le script va s'interrompre comme d'habitude."

On peut penser que peu importe la fonction qu'on écrit, si cette fonction appelle un callback (ou plusieurs), elle sera exécutée de manière asynchrone. **Cela est faux** (ex :

```
//définition de la fonction
function callbackHandler(monCallback){
  console.log(`après cette instruction, j'appelle monCallback`);
  monCallback();
}
```

//appel de la fonction
`callbackHandler(function(){console.log(`j'ai fini !`);});`. Bien que `callbackHandler()` ait un callback comme paramètre, elle sera exécutée de manière synchrone. En effet, pour qu'une fonction s'exécute de manière asynchrone, elle doit utiliser en elle des fonctions natives de JavaScript qui sont par défaut asynchrones. Dans les navigateurs, ces fonctions font partie des **Web APIs** (ex : `setTimeout()` ou `fetch()`). Dans Node.js, presque toutes les fonctions de son API sont asynchrones.

Donc, dorénavant, quand vous codez, demandez-vous : « Est-ce que la tâche que je veux implémenter peut s'effectuer en arrière-plan ? (Surtout s'il s'agit d'une tâche qui prend beaucoup de temps comme un minuteur, une requête http, un accès au stockage, etc.) », si oui, trouvez un moyen de l'implémenter de manière asynchrone en vous renseignant sur les Web APIs (via **MDN Web Docs, par exemple) ou les API côté-serveur, si vous travaillez côté-serveur, qui peuvent vous permettre d'y arriver. Sinon, implémentez-le de manière synchrone.**

LES PROMESSES

Pour résoudre le problème des callbacks imbriqués, les **promesses** ont été introduites. Une promesse est un objet, instance de la classe **Promise** qui représente une valeur qui sera disponible maintenant ou dans le futur. Une promesse utilise deux méthodes principales (il en a plusieurs autres) :

- **then()** : pour gérer une promesse réussie (pour en savoir plus sur elle, voir **MDN Web Docs**).
- **catch()** : pour gérer une promesse échouée (pour en savoir plus sur elle, voir **MDN Web Docs**).

Le constructeur de la promesse retournée prend un argument. Cet argument doit impérativement respecter les règles suivantes :

- Cet argument doit s'agir d'une fonction ;
- Cette fonction doit avoir deux paramètres, le premier censé contenir une fonction résolvant la promesse (on le dénote souvent `resolve()`, mais on peut le nommer comme on veut), l'autre censé contenir un message indiquant l'erreur (on le dénote souvent `reject()`, mais on peut aussi le nommer comme on veut).

(ex :

```

function chargerDonnées(url){ //callbackHandler défini avec une promesse
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const données = `Données chargées depuis ${url}`;
            if (données){
                resolve(données);
            } else{
                reject('Erreur lors du chargement des données');
            }
        }, 2000);
    });
}

chargerDonnées('https://api.exemple.com') //appel de la fonction
    .then((données) => {
        console.log(données); //affiche : Données chargées
    })
    .catch((erreur) => {
        console.error(erreur);
    });
). Ce code fait la même chose que celui avec la notation précédente, mais elle plus est simplifiée.

```

Les promesses améliorent la lisibilité du code et permettent d'enchaîner les opérations asynchrones plus facilement avec `then()`. Elles permettent également de gérer les erreurs plus efficacement avec `catch()`.

Pour mieux comprendre la raison, essayez de visualiser le code suivant :

```

function Operation1(successCallback, failureCallback){ //callbackHandler1
    setTimeout(() => {
        console.log("Première opération");
        const success = Math.random() > 0.5; //Simulation d'une réussite ou d'un échec
        if (success){
            successCallback("Première opération réussie !");
        } else{
            failureCallback("Première opération échouée.");
        }
    }, 1000);
}

function Operation2(successCallback, failureCallback){ //callbackHandler2
    setTimeout(() => {
        console.log("Deuxième opération");
        const success = Math.random() > 0.5;
        if (success){
            successCallback("Deuxième opération réussie !");
        } else{
            failureCallback("Deuxième opération échouée.");
        }
    }, 1000);
}

function Operation3(successCallback, failureCallback){ //callbackHandler3
    setTimeout(() => {
        console.log("Troisième opération");
        const success = Math.random() > 0.5;
        if (success){
            successCallback("Troisième opération réussie !");
        } else{
            failureCallback("Troisième opération échouée.");
        }
    }, 1000);
}

```

```
//Enchaînement des callbacks imbriqués
Operation1(
  (result1) => {
    console.log(resultat1);
    Operation2(
      (resultat2) => {
        console.log(resultat2);
        Operation3(
          (resultat3) => {
            console.log(resultat3);
          }, (erreur3) => {console.error(erreur3);});
        }, (erreur2) => {console.error(erreur2);});
      }, (erreur1) => {console.error(erreur1);});
```

`Operation1()`, `Operation2()`, et `Operation3()` sont des fonctions asynchrones simulées avec `setTimeout`. Chaque opération réussit ou échoue de manière aléatoire (en utilisant `Math.random()`). Les callbacks sont imbriqués : si `Operation1()` réussit, alors on appelle `Operation2()`, ainsi de suite jusqu'à `Operation3()`. En cas d'échec à n'importe quelle étape, le callback d'erreur correspondant est exécuté. Si vous avez compris comment cet enchainement fonctionne en moins de 30 secondes, félicitations ! Mais sachez que là, il n'y avait que 3 fonctions (imaginez s'il y en avait 10 !). Réécrire le même code avec des promesses simplifie les choses :

```
function Operation1(){ //callbackHandler1
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Première opération");
      const success = Math.random() > 0.5;
      if (success){
        resolve("Première opération réussie !");
      } else{
        reject("Première opération échouée.");
      }
    }, 1000);
  });
}

function Operation2(){ //callbackHandler2
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Deuxième opération");
      const success = Math.random() > 0.5;
      if (success){
        resolve("Deuxième opération réussie !");
      } else{
        reject("Deuxième opération échouée.");
      }
    }, 1000);
  });
}

function Operation3(){ //callbackHandler3
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      console.log("Troisième opération");
      const success = Math.random() > 0.5;
      if (success){
        resolve("Troisième opération réussie !");
      } else{
        reject("Troisième opération échouée.");
      }
    }, 1000);
  });
}
```

```

        }, 1000);
    });
}

//Enchaînement des promesses, plus lisible que celui des callbacks
Operation1().then((resultat1) => {
    console.log(resultat1);
    return Operation2(); //La valeur renournée peut être utilisée au then() suivant
}).then((resultat2) => {
    console.log(resultat2);
    return Operation3();
}).then((resultat3) => {
    console.log(resultat3);
}).catch((erreur) => {
    console.error(erreur);
});
}

```

Chaque fonction (`Operation1()`, `Operation2()`, `Operation3()`) retourne maintenant une promesse (Promise). Si l'opération réussit, on appelle `resolve()` pour continuer avec le `then()`. En cas d'échec, on appelle `reject()` pour déclencher le `catch()`. L'enchaînement des opérations est beaucoup plus lisible avec les promesses. Chaque appel `then()` attend que la promesse précédente soit résolue avant de continuer, et le catch gère toutes les erreurs éventuelles.

Avec l'introduction des mots-clés `async` et `await` en ES2017, la gestion des promesses est devenue encore plus simple. Une fonction déclarée avec le mot-clé `async` retourne toujours une promesse. Le mot-clé `await` est utilisé à l'intérieur de cette fonction pour attendre la résolution d'une promesse avant de continuer l'exécution du code. On utilise ces deux expressions TOUJOURS ensemble (ex :

```

function chargerDonnées(url){ //callbackHandler défini avec une promesse
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            const données = `Données chargées depuis ${url}`;
            if (données){
                resolve(données);
            } else{
                reject('Erreur lors du chargement des données');
            }
        }, 2000);
    });
}

async function chargerToutesLesDonnées(){ //on utilise async slmnt sur une fonction
try{
    const données1 = await chargerDonnées('https://api.exemple.com');
    console.log(données1); //équivalent à un then()

    const données2 = await chargerDonnées('https://api.autre.com');
    console.log(données2); //équivalent au then() suivant dans la chaîne
} catch(erreur){
    console.error(erreur);
}
}

//appel de la fonction asynchrone
chargerToutesLesDonnées(); Si au moins un chargerDonnées() échoue, une exception est lancée.

```

Avantages de `async/await` :

1. Le code ressemble à du code synchrone, ce qui le rend plus facile à comprendre.
2. L'utilisation de `try/catch` permet une gestion des erreurs plus simple et cohérente dans l'ensemble du code.

3. Plus besoin d'enchaîner les `then()` manuellement. Donc, le code :

```

function Operation1(){ //callbackHandler1
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Première opération");
            const success = Math.random() > 0.5;
            if (success){
                resolve("Première opération réussie !");
            } else{
                reject("Première opération échouée.");
            }
        }, 1000);
    });
}

function Operation2(){ //callbackHandler2
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Deuxième opération");
            const success = Math.random() > 0.5;
            if (success){
                resolve("Deuxième opération réussie !");
            } else{
                reject("Deuxième opération échouée.");
            }
        }, 1000);
    });
}

function Operation3(){ //callbackHandler3
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Troisième opération");
            const success = Math.random() > 0.5;
            if (success){
                resolve("Troisième opération réussie !");
            } else{
                reject("Troisième opération échouée.");
            }
        }, 1000);
    });
}

//Enchaînement des promesses
Operation1().then((resultat1) => {
    console.log(resultat1);
    return Operation2();
}).then((resultat2) => {
    console.log(resultat2);
    return Operation3();
}).then((resultat3) => {
    console.log(resultat3);
}).catch((erreur) => {
    console.error(erreur);
});

S'écrira dorénavant :
function Operation1(){ //callbackHandler1
    return new Promise((resolve, reject) => {
        setTimeout(() => {

```

```

        console.log("Première opération");
        const success = Math.random() > 0.5;
        if (success){
            resolve("Première opération réussie !");
        } else{
            reject("Première opération échouée.");
        }
    }, 1000);
}

function Operation2(){ //callbackHandler2
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Deuxième opération");
            const success = Math.random() > 0.5;
            if (success){
                resolve("Deuxième opération réussie !");
            } else{
                reject("Deuxième opération échouée.");
            }
        }, 1000);
    });
}

function Operation3(){ //callbackHandler3
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            console.log("Troisième opération");
            const success = Math.random() > 0.5;
            if (success){
                resolve("Troisième opération réussie !");
            } else{
                reject("Troisième opération échouée.");
            }
        }, 1000);
    });
}

async function enchaînerLesOpérations(){ //Enchaînement
    try{
        const résultat1 = await Operation1();
        console.log(résultat1); //équivalent au premier then()

        const résultat2 = await Operation2();
        console.log(résultat2); //équivalent au deuxième then()

        const résultat3 = await Operation3();
        console.log(résultat3); //équivalent au troisième then()
    } catch(erreur){
        console.error(erreur); //équivalent au catch()
    }
}

enchaînerLesOpérations();

```

Vu que la notation `async/await` rend le code plus proche du style synchrone, on peut à présent parcourir les données que nous obtenons de manière asynchrone. On le dit aussi : « Itération asynchrone ». Elle utilise une forme spéciale de la boucle `for` : La boucle `for/await` (voir *JavaScript : The definitive guide*).

CHAPITRE 12 : JAVASCRIPT DANS LE NAVIGATEUR WEB

JavaScript a été créé initialement dans le but de permettre d'écrire des pages Web dynamiques. Il s'est développé au fil du temps, et aujourd'hui, on l'utilise même en dehors du navigateur. On l'utilise du côté serveur grâce à Node.Js, du côté mobile grâce à React Native, du côté Desktop grâce à Electron.Js, etc. Ce chapitre parle de l'utilisation de JavaScript dans les navigateurs Web.

Un navigateur Web est un logiciel qui permet principalement d'afficher des pages ou documents Web. Ces pages Web sont échangées entre navigateurs et serveurs via le réseau sous un format de description appelé HTML. Ce format dicte au navigateur, en plus de la structure de la page, de ce qu'il faut d'autre pour qu'il puisse la reconstruire fidèlement, notamment, les médias, les styles et les scripts. Donc inclure du JavaScript dans du code HTML est une des phases par laquelle on peut être emmené à passer lorsqu'on écrit le code d'une page Web. On peut inclure un script JavaScript dans une page HTML de trois façons :

1. **Inclusion en ligne** : Elle consiste à placer son script directement dans un élément HTML, dans un fichier d'extension « .html », en l'affectant comme valeur à un attribut d'évènement rattaché à cet élément (ex : `<button onclick="alert('coucou!');">appuie-moi !</button>`). En gros, ça signifie : « quand l'utilisateur clique sur ce bouton, affiche le message "coucou !" dans un popup ». On utilise l'inclusion en ligne lorsqu'on veut écrire un script pour un élément précis de la page.
2. **Inclusion interne** : Elle consiste à placer son script dans un élément `<script>`, dans le `<head>` ou dans le `<body>` d'un fichier d'extension « .html ». On peut y écrire des scripts impactant la page entière (ex :

Code HTML

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>monDocumentHTML</title>
</head>
<body>
  <script>
    console.log('yo dude!');
  </script>
</body>
</html>.
```

Il est peu conseillé d'utiliser l'inclusion interne car du fait qu'on inclut du code JavaScript directement dans un fichier « .html » qui contient déjà du HTML, il peut être facteur de confusions.

Quand on met l'élément `<script>` dans le `<head>`, cela peut ralentir le chargement de la page. En effet, vu que le HTML est lu de haut en bas, si le script qu'on rajoute est relativement long, le navigateur peut s'attarder à exécuter le script avant de continuer le traitement du code HTML et donc, l'empêcher de s'afficher. D'où, il est fortement déconseillé de le faire, sauf en cas nécessaire.

Quand on met l'élément `<script>` dans le `<body>`, Il est conseillé de le mettre tout en bas, en dernier. En effet, vu que le HTML est lu de haut en bas, lorsque le navigateur en arrivera au niveau d'exécuter le script, il serait mieux que la page soit déjà construite et affichée, et surtout pour le **DOM**, utile pour manipuler la page dynamiquement via JavaScript.

On peut ajouter un autre attribut, au choix, à l'élément `<script>` : `async` ou `defer`. Lorsqu'on rajoute `async`, le script est exécuté en arrière-plan pendant que le reste de la page est traitée, peu importe l'endroit où cette balise se trouve (ex :

```
<script async>
  console.log('yo dude!');
</script>.
```

Lorsqu'on rajoute `defer`, le script est exécuté uniquement après que tout le HTML de la page soit traitée, peu importe l'endroit où cette balise se trouve (ex :

```
<script defer>
  console.log('yo dude!');
</script>.
```

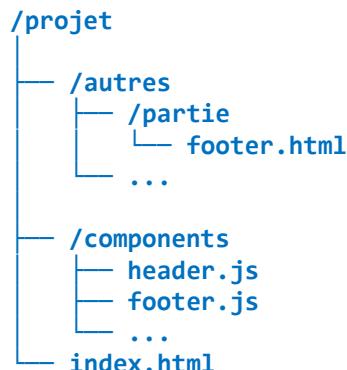
A mon avis, je vous conseillerai de n'utiliser aucun de ces attributs et de simplement mettre cet élément à la fin de votre `<body>`.

3. **Inclusion externe** : Elle consiste à écrire du JavaScript dans un fichier à part, d'extension « .js », et à l'inclure dans le fichier HTML en renseignant son *chemin* (relatif ou absolu) à l'attribut `src` de l'élément `<script>`, dans le `<head>` ou dans le `<body>` de ce fichier (ex :

Code HTML

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>monDocumentHTML</title>
</head>
<body>
  <script src="./hello.js"></script>
</body>
</html>). Cette notation est à préférer sur toutes les autres. Il faut juste s'assurer que la balise <script> se trouve tout en bas dans le <body>, pour ne pas bloquer le html. Idem, on peut utiliser ici aussi les attributs async ou defer.
```

Le chemin renseigné à l'attribut `src` peut être absolu ou relatif. Un chemin absolu et une notation décrivant tous les dossiers qu'on doit parcourir, en partant de la racine du système de fichier de l'ordinateur (ex : `C:/`) pour atteindre une ressource. Cette notation est utile pour décrire les chemins complexes ou les URLs (ex : `C:/Users/DELL/Documents/AUTRES/helloworld.html`, `https://github.com/SwayIchvalan`, etc.). Un chemin relatif est une notation décrivant les dossiers qu'il faut parcourir, en partant de l'emplacement du fichier actuel, pour atteindre une ressource. Pour écrire le chemin relatif d'une ressource qui se trouve dans le même dossier que celui du fichier dans lequel on se trouve, on écrit juste le nom du fichier ou bien le nom précédé de « ./ » (ex : `Oops.js`, `./moi_beau_et_gentil.png`, etc.). On utilise « ../ » pour voyager dans les dossiers parent. Pour aller dans les dossiers enfants, on écrit le chemin absolu en prenant le nom du dossier où se trouve le fichier actuel comme racine (ex :



Supposons qu'on est dans `index.html` qui est dans le dossier `projet`. Si on veut spécifier un chemin relatif vers `header.js`, qui est dans le dossier `components`, un dossier enfant à `projet`, on écrit :

`projet/components/header.js`.

Supposons maintenant qu'on est dans `footer.html` qui est dans le dossier `partie`, lui-même se trouvant dans le dossier `autres`, lui-même se trouvant dans le dossier `projet`. Si on veut spécifier un chemin relatif vers `header.js`, qui est dans le dossier `components`, on écrit : `../../components/header.js`. On remonte vers le dossier `autres`, puis vers le dossier `projet`, on entre dans le dossier `components` et on accède à `header.js`.

La programmation en JavaScript dans le navigateur est asynchrone, non-bloquant et orienté-événements, ça veut dire qu'en JavaScript (dans le navigateur), toutes les fonctions des API sont exécutées dans la phase asynchrone. Ensuite, une fois que la page est affichée, toute interaction entre vous et le navigateur peut être capturée comme un événement (ex : clic sur un bouton, survol d'une zone de la page, etc.) et donc, tout ce que vous avez à faire, c'est spécifier comment le navigateur doit se comporter à chaque fois qu'il en capture un. Les navigateurs mettent à notre disposition ces 3 capacités grâce aux **Web APIs**. Il en existe beaucoup, mais dans ce chapitre, je vais en introduire uniquement trois que je trouve fondamentales : **l'API DOM**, **l'API Fetch** et le **Web Storage API**. Le reste, vous pouvez aller sur **MDN Web Docs** pour découvrir.

L'API DOM (DOCUMENT OBJECT MODEL)

Cet API nous donne le contrôle sur la structure HTML de notre page Web. Après qu'un fichier HTML soit lu et la page y correspondante affichée, le navigateur construit un objet JavaScript dont les propriétés sont d'autres objets, appelés **nœuds**, qui reconstituent la structure complète de cet HTML. Il nous l'expose ensuite à travers une API qui porte le nom : **DOM**, et le stocke dans l'objet global. Programmer avec l'API DOM consiste généralement à capturer les événements qui se produisent sur ces nœuds et en scripter un comportement. Voici une petite terminologie concernant les événements :

- Chaque évènement a un nom (**event type**). Par exemple, `'load'` ;
- Chaque évènement est rattaché à un nœud (**event target**), représentant un élément HTML se trouvant dans la page sur laquelle agit le script ;
- Chaque évènement est capturé, puis géré à travers une fonction appelée **event handler** ou **event listener** par le navigateur. Lorsque le navigateur invoque un event listener sur un event target, on dit que le navigateur *déclenche* ou *propage* un évènement ;
- Chaque nœud a des propriétés qui décrivent les événements qui peuvent lui être associés. Ces propriétés sont des objets qui contiennent chacun, les détails associés à l'évènement à laquelle ils sont liés (le nom, le target, etc.). Par exemple, le nœud `<button>` peut avoir les propriétés `onclick`, `onsubmit`, représentant les événements `'click'`, `'submit'` respectivement.

Il y a deux manières de définir un comportement à adopter en réaction à un évènement. La première consiste à affecter une fonction à la propriété d'un nœud qui représente l'évènement. Toutes les propriétés représentant les événements d'un nœud ont « `on-` » comme préfixe et l'`event type` comme suffixe (ex :

```
window.onload = function (){
```

```
    console.log('fenêtre ouverte');
```

`};`). Le nœud `window` représente la fenêtre du navigateur. La propriété `onload` est l'objet qui est lié à l'évènement `'load'`, pour lequel on veut que le navigateur exécute la fonction qu'on lui affecte. En gros, ça signifie : « lorsque la fenêtre est ouverte, afficher "fenêtre ouverte" sur la console ».

Une autre variante de cette manière consiste à définir un comportement directement dans une balise HTML, dans un fichier HTML (ex : `<button onclick="console.log('ça pète ?!');">ok!</button>`). Il suffit de rajouter un attribut HTML dont le nom correspond à la propriété d'un nœud qui représente l'évènement et de lui affecter comme valeur, une chaîne contenant tout le code JavaScript qu'on veut exécuter si cet évènement est déclenchée (chaque élément à ses attributs d'évènement, pour en savoir plus, voir sur [MDN Web Docs](#)).

La deuxième manière consiste à utiliser une méthode `addEventListener()` sur le nœud pour lequel on veut spécifier un comportement en cas d'un évènement se produisant sur lui (ex :

Code HTML

```
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>monDocumentHTML</title>
</head>
<body>
    <button id="monBouton">Clique-moi</button>
    <script>
        let b = document.querySelector("#monBouton");
        b.addEventListener("click", () =>{console.log("Arigato!");}, {capture: true,
                                                               once: false,
                                                               passive: true});
    </script>
</body>
</html>
```

`</html>`). La fonction prend trois arguments, le premier est une chaîne de caractères qui représente le event-type, à ne pas confondre avec l'objet associé à l'évènement (ex : `onclick`). Le deuxième est une fonction callback qui est invoquée lorsque l'évènement correspondant à l'`event type` se produit sur le nœud associé. Le troisième est optionnel. Il s'agit d'un objet de trois propriétés qui prennent chacune une valeur

booléenne : `capture`, `once` et `passive`. Lorsque `capture` a comme valeur `true` (sa valeur par défaut), cela veut dire que la fonction sera appelée par le navigateur comme event handler. Lorsque `once` a comme valeur `true`, cela veut dire que le navigateur ne va appeler cette fonction qu'une fois. Sa valeur par défaut est `false`. Lorsque `passive` a comme valeur `true`, cela veut dire que l'event handler ne va jamais invoquer la méthode `preventDefault()` qui empêche le comportement par défaut du navigateur lorsque le même évènement se produit.

On utilise la méthode `removeEventListener()` pour supprimer un event handler précédemment créé. Il prend les mêmes arguments.

Toutes les fonctionnalités de l'API sont stockées comme propriétés dans l'objet `document`. Pour définir un nœud qui manipule les éléments HTML d'une page Web, on doit utiliser des fonctions de sélection d'éléments de l'objet `document`. Il y en a 6 :

1. `document.querySelector()` : sélectionne le 1^{er} élément correspondant au sélecteur CSS fourni. Si aucun élément ne correspond, il retourne `null` (ex :

```
//Sélectionne le premier <div> de la page html associée
let firstDiv = document.querySelector('div');
```

```
//Sélectionne le premier élément avec la classe 'my-class'
let specificClass = document.querySelector('.my-class');
```

```
//Sélectionne l'élément avec l'ID 'my-id'
let specificId = document.querySelector('#my-id');
```

//Sélectionne le premier <p> qui est un enfant direct de <div>
let child = document.querySelector('div>p'); Il est recommandé de préférer celle-ci sur toutes les autres fonctions lorsqu'on veut manipuler un seul élément à la fois.

2. `document.querySelectorAll()` : retourne une `NodeList` de tous les éléments correspondant au sélecteur CSS fourni. La `NodeList` est une collection de nœuds qui est mise à jour en temps réel avec le DOM (ex :

```
//Retourne une NodeList (tableau) de tous les <div>
let allDivs = document.querySelectorAll('div');
```

```
//Retourne un tableau de tous les <p> avec la classe 'my-class'
let allParagraphs = document.querySelectorAll('p.my-class');
```

*//Retourne tous les qui sont enfants directs de *
let allItems = document.querySelectorAll('ul > li'); La NodeList est itérable, ce qui permet d'utiliser des méthodes comme `forEach()` pour les parcourir. Il est recommandé de préférer celle-ci sur toutes les autres fonctions lorsqu'on veut manipuler plusieurs éléments à la fois.

3. `document.getElementsByName()` : sélectionne tous les éléments ayant un attribut `name` avec la valeur spécifiée. Cette méthode est souvent utilisée pour les formulaires HTML (ex :

```
//Retourne une HTMLCollection (tableau) de tous les éléments avec name='username'
let inputs = document.getElementsByName('username')); Elle n'est pas directement itérable avec la méthode forEach().
```

4. `document.getElementsByTagName()` : sélectionne tous les éléments ayant le nom de balise spécifié. Cette méthode retourne un `HTMLCollection` (ex :

```
//Retourne une HTMLCollection (tableau) de tous les <div>
let divs = document.getElementsByTagName('div');
let paragraphes = document.getElementsByTagName('p'); //Retourne tous les <p>).
```

5. `document.getElementsByClassName()` : sélectionne tous les éléments ayant la ou les classes spécifiées. Retourne un `HTMLCollection` (ex :

```
//Retourne une HTMLCollection (tableau) de tous les éléments avec la classe 'item'
let items = document.getElementsByClassName('item');
```

6. `document.getElementById()` : sélectionne l'élément ayant l'ID spécifié. L'ID doit être unique dans le document (ex :

```
//Retourne l'élément avec l'ID 'main-header'
let header = document.getElementById('main-header'));
```

Un document HTML est structuré comme une arborescence de nœuds, avec chaque nœud représentant un élément, un attribut ou du texte. Voici un exemple de structure de document HTML :

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exemple</title>
</head>
<body>
  <header id="main-header">
    <h1>Mon site</h1>
  </header>
  <main>
    <section>
      <h2>À propos</h2>
      <p class="intro">Bienvenue dans mon site!</p>
    </section>
    <section>
      <h2>Contact</h2>
      <form>
        <input type="text" name="username" placeholder="entre ton username">
        <button type="submit">Soumettre</button>
      </form>
    </section>
  </main>
  <footer>
    <p>&copy; 2024 mon site</p>
  </footer>
</body>
</html>
```

Traverser le DOM consiste à naviguer à travers les nœuds enfants, frères et parents. Voici quelques champs utiles définies dans l'objet document qui permettent de naviguer à travers le DOM :

1. **parentNode** : Retourne le parent du nœud actuel (le nœud qui contient le nœud actuel) (ex :
`let header = document.getElementById('main-header');`
`let parent = header.parentNode; //Retourne <body>.`
2. **childNodes** : Retourne une liste de tous les nœuds enfants (inclut les nœuds de texte et les espaces blancs) (ex :
`let enfants = document.getElementById('main-header').childNodes;`)
3. **firstChild** et **lastChild** : Retourne respectivement le premier et le dernier enfant du nœud (ex :
`let premierEnfant = document.getElementById('main-header').firstChild;`
`let dernierEnfant = document.getElementById('main-header').lastChild;`)
4. **nextSibling** et **previousSibling** : Retourne respectivement le nœud frère suivant et précédent (ex :
`let suiv = document.getElementById('main-header').nextSibling;`
`let prev = document.getElementById('main-header').previousSibling;`)
5. **children** : Retourne une HTMLCollection des enfants qui sont des éléments(ex :
`let elementsEnfant = document.getElementById('main-header').children;`)
6. **firstElementChild** et **lastElementChild** : Retourne respectivement le premier et le dernier enfant qui est un élément (ex :
`let firstElement = document.getElementById('main-header').firstElementChild;`
`let lastElement = document.getElementById('main-header').lastElementChild;`)

On peut aussi, à la place de manipuler des nœuds, manipuler juste leurs attributs (ex : `name`, `id`, etc.). Cela se fait à l'aide des méthodes suivantes :

1. **Lire un attribut** : On lit la valeur d'un attribut HTML grâce à la fonction `getAttribute('attribut')` (ex :
`let source = document.querySelector('img').getAttribute('src');`)
2. **Modifier un attribut** : On utilise la méthode `setAttribute('attribut', 'valeur')` (ex :
`document.querySelector('img').setAttribute('src', './new-image.jpg');`)

3. **Supprimer un attribut d'un élément** : On utilise la méthode `removeAttribute('attribut')` (ex : `document.querySelector('img').removeAttribute('src');`).

On peut aussi manipuler les contenus de nos éléments HTML :

1. **Créer un nouvel élément HTML** : On utilise la fonction `createElement('élément')` (ex : `let nouvelElement = document.createElement('div');`).
2. **Ajouter du contenu textuel à un élément** : On affecte le texte qu'on veut insérer dans l'élément au champ `textContent` (ex : `nouvelElement.textContent = 'Hello, world!';`).
3. **Insérer un élément HTML dans un autre** : On utilise la fonction `appendChild()` (ex : `document.body.appendChild(nouvelElement);`).
4. **Supprimer un élément du DOM** : On utilise la fonction `removeChild()` (ex : `document.body.removeChild(nouvelElement);`).
5. **Remplacer le contenu HTML dans un élément** : On affecte le code HTML à insérer dans l'élément comme un string au champ `innerHTML` (ex : `let monElement = document.querySelector('.my-class');`
`monElement.innerHTML = 'Bold text';`).

En utilisant ces méthodes et concepts, vous pouvez naviguer et manipuler efficacement la structure des documents HTML en JavaScript, créant ainsi des interfaces dynamiques et interactives.

De même qu'avec le DOM, on peut manipuler du CSS via une extension de l'API DOM : le **CSSOM** (CSS Object Model). C'est une API puissante qui nous permet de modifier dynamiquement l'apparence des éléments d'une page web. En combinant JavaScript et CSS, il est possible de modifier les styles des éléments en réponse à des événements ou en fonction des interactions des utilisateurs.

On peut modifier le style en utilisant les propriétés du champ `style` : La manière la plus directe de modifier les styles CSS consiste à utiliser le champ `style` des nœuds du DOM. Cette propriété permet de définir ou de modifier directement des propriétés CSS spécifiques (ex :

```
monElement.style.color = 'red'; //Changer la couleur du texte en rouge
monElement.style.fontSize = '20px'; //Changer la taille de la police).
```

Cependant, cette méthode ne modifie que le *style en ligne* de l'élément, c'est-à-dire le style directement appliqué à l'élément lui-même, et non ceux définis dans des feuilles de style externes.

On peut utiliser les propriétés du champ `classList` : Il est souvent plus efficace et plus flexible de gérer les classes CSS (ex : `.intro`) au lieu de modifier directement les propriétés individuelles. La propriété `classList` permet de manipuler les classes appliquées à un élément. Avec elle, on peut ajouter, supprimer ou basculer des classes CSS. Voici quelques méthodes clés de `classList` :

1. `add('nomClasse')` : ajoute une ou plusieurs classes CSS à un nœud (ex : `monElement.classList.add('new-class');` //Ajoute une classe).
2. `remove('nomClasse')` : supprime une ou plusieurs classes CSS à un nœud (ex : `monElement.classList.remove('old-class');` //Supprime une classe).
3. `toggle('nomClasse')` : ajoute la classe CSS si elle n'existe pas, la supprime si elle est présente (ex : `monElement.classList.toggle('active');` //Bascule une classe).
4. `contains('nomClasse')` : vérifie si l'élément possède une classe spécifique (ex : `monElement.classList.contains('active');` //Retourne true ou false).

L'avantage de cette méthode est que les classes permettent de regrouper plusieurs propriétés CSS sous un seul nom, ce qui facilite la gestion des styles complexes.

On peut modifier les feuilles de style directement : Il est également possible de modifier ou d'ajouter des règles CSS globales à partir de JavaScript. Pour ce faire, on peut accéder aux feuilles de style d'un document via l'objet `document.styleSheets`, puis manipuler les règles CSS.

1. `insertRule('règleCSS', index(optionnel))` : ajoute une nouvelle règle CSS dans la feuille de style spécifiée (ex : `let x = document.styleSheets[0]; //Accède à la 1re feuille incluse dans le HTML
x.insertRule('body { background-color: blue; }', x.cssRules.length);`). L'`index` indique juste l'endroit où la règle doit être rajoutée dans la feuille de style. Dans le cas de cet exemple, elle est rajoutée à la dernière position de la première feuille.
2. `deleteRule(index)` : supprime une nouvelle règle CSS dans la feuille de style spécifiée (ex : `feuille.deleteRule(0); //Supprime la 1re feuille de style incluse`).

On peut aussi lire les styles calculés avec `getComputedStyle(noeud)` : Pour obtenir le style *effectif* d'un élément, incluant ceux définis dans des feuilles de style externes ou hérités. Cette méthode renvoie un objet qui contient toutes les propriétés CSS d'un élément tel qu'elles sont appliquées après évaluation (ex : `let monstyle = window.getComputedStyle(monbouton); console.log(monstyle.backgroundColor); //Affiche la couleur de fond calculée`). Cela est particulièrement utile pour récupérer les valeurs des styles qui ne sont pas directement accessibles via `style`.

On peut aussi manipuler des pseudo-classes CSS. En effet, les pseudo-classes comme `::before` et `::after` ne peuvent pas être directement manipulées via `style`. Toutefois, avec `getComputedStyle()`, il est possible de lire les styles de ces pseudo-classes (ex :

```
let pseudoElementStyle = window.getComputedStyle(element, '::before');
console.log(pseudoElementStyle.content); //Affiche le contenu du pseudo-élément ::before
```

Cependant, pour modifier ces pseudo-éléments, il est nécessaire d'agir sur les règles CSS dans la feuille de style.

Enfin, on peut manipuler les styles en fonction d'événements. Le CSS peut être modifié dynamiquement en réponse à des événements utilisateur, tels que des clics ou des survols de souris. Par exemple, en utilisant des écouteurs d'événements, on peut déclencher des changements de style lors d'interactions spécifiques (ex :

```
monElement.addEventListener('click', function (){
  monElement.style.backgroundColor = 'yellow'; //Change le fond lors d'un clic
});
```

Cela permet de rendre une interface web interactive et réactive aux actions des utilisateurs.

Voici un exemple concret d'un bouton qui active/désactive le mode nuit sur une page web :

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <style>
    body{
      background-color: white;
      color: black;
    }
    body.mode-nuit{
      background-color: black;
      color: white;
    }
  </style>
  <title>monDocumentHTML</title>
</head>
<body>
  <button id="monInterruuteur">Activer/Désactiver le mode nuit</button>
  <script>
    //Le rendre persistant grâce à une fonction de l'API Web Storage
    let modeNuit = localStorage.getItem("modeNuit");

    if (modeNuit === "activé"){
      document.body.classList.add("mode-nuit");
    }
    document.getElementById("monInterruuteur").addEventListener("click", function(){
      document.body.classList.toggle("mode-nuit");
      if (document.body.classList.contains("mode-nuit")){
        localStorage.setItem("modeNuit", "activé");
      } else{
        localStorage.setItem("modeNuit", "désactivé");
      }
    });
  </script>

```

```
</script>
</body>
</html>).
```

PAUSE : LE PROTOCOLE HTTP

Le protocole **HTTP** (Hypertext Transfer Protocol) est un protocole de communication essentiel pour échanger des données sur le Web. Il permet aux clients (comme les navigateurs) de demander des ressources (pages, images, vidéos) auprès des serveurs. Bien qu'initiallement conçu pour le transfert de documents HTML, il prend également en charge de nombreux autres types de fichiers. Il est à la base du fonctionnement du Web moderne, permettant à un large éventail de services et d'applications de fonctionner.

HTTP est un protocole *stateless*, ce qui signifie qu'il ne conserve pas d'informations d'une requête à l'autre. Cela signifie que chaque interaction entre un client et un serveur est indépendante. Toutefois, des mécanismes comme les cookies et les sessions sont utilisés pour maintenir l'état entre plusieurs requêtes successives.

Une *requête HTTP* comporte les éléments suivants :

1. Ligne de requête : Spécifie la méthode HTTP (comme GET ou POST), l'URL demandée, et la version du protocole HTTP.
2. En-têtes de requête : Transmettent des informations supplémentaires telles que le type de contenu ou les données d'authentification.
3. Corps de la requête (facultatif) : Peut contenir des données, notamment lors de requêtes POST ou PUT.

Une *réponse HTTP* contient trois parties principales :

1. Ligne d'état : Inclut la version HTTP, le code d'état (comme 200 OK), et une brève explication.
2. En-têtes de réponse : Indiquent des informations sur la ressource, comme son type (Content-Type) ou sa taille.
3. Corps de la réponse : Contient la ressource demandée (page HTML, fichier JSON, image, etc.).

Le **caching HTTP** est un mécanisme permettant de stocker temporairement une copie d'une ressource pour améliorer les performances réseau et réduire la charge serveur. Il joue un rôle crucial dans l'optimisation des échanges sur le Web en permettant de répondre aux requêtes sans avoir à solliciter à chaque fois le serveur d'origine. Voici un aperçu détaillé du fonctionnement du caching HTTP.

L'objectif du caching est de réduire :

1. La latence : Les utilisateurs obtiennent plus rapidement les ressources demandées, car celles-ci sont servies depuis un cache local (navigateur ou proxy) plutôt que de traverser le réseau pour interroger le serveur d'origine.
2. Le trafic réseau : Moins de requêtes sont envoyées au serveur, ce qui allège la charge réseau.
3. La charge serveur : Le serveur est sollicité moins souvent, ce qui permet de libérer des ressources pour d'autres tâches ou utilisateurs.
4. Les coûts : En limitant les allers-retours avec le serveur, on réduit l'utilisation de la bande passante.

Il existe différents types de caches dans le cadre des requêtes HTTP :

1. Cache du navigateur : Stocke des ressources localement sur l'ordinateur de l'utilisateur. Les navigateurs mettent en cache les fichiers comme les images, CSS, et JavaScript afin d'améliorer la rapidité de chargement des pages visitées.
2. Cache intermédiaire ou proxy : Situé entre le client et le serveur, ce cache est utilisé par des entités telles que des fournisseurs d'accès à Internet pour stocker et servir des ressources fréquemment demandées par plusieurs utilisateurs.
3. Cache côté serveur : Les serveurs peuvent eux aussi mettre en cache certaines données pour accélérer les traitements des requêtes répétées, souvent avec des systèmes comme Redis ou Varnish.

Les requêtes et les réponses HTTP utilisent plusieurs en-têtes pour gérer le comportement du cache.

1. **Cache-Control** : Cet en-tête est le plus utilisé pour spécifier les directives de cache, tant côté client que serveur. Il peut inclure des directives comme :

- no-cache : Le client doit valider la réponse avec le serveur avant de l'utiliser.
 - no-store : Ne pas stocker la réponse dans aucun cache.
 - max-age : Durée maximale en secondes pendant laquelle une réponse peut être mise en cache (par exemple, max-age=3600 pour une heure).
 - public : La réponse peut être mise en cache par n'importe quel cache intermédiaire.
 - private : La réponse est spécifique à un utilisateur et ne doit pas être mise en cache par des caches partagés.
2. **Expires** : Cet en-tête fournit une date et une heure après lesquelles la réponse est considérée comme obsolète. Il est généralement remplacé par Cache-Control: max-age mais reste compatible pour les anciens systèmes.
 3. **ETag (Entity Tag)** : Cet en-tête fournit un identifiant unique pour une version spécifique d'une ressource. Si la ressource change, l'ETag change également. Cela permet au cache de valider une ressource avec le serveur (via une requête If-None-Match), ce qui permet de savoir si la ressource a été modifiée.
 4. **Last-Modified** : Indique la dernière date à laquelle la ressource a été modifiée. Lors d'une nouvelle requête, le client peut utiliser If-Modified-Since pour demander uniquement les ressources modifiées après une certaine date.
 5. **Vary** : Cet en-tête informe les caches de prendre en compte certains en-têtes lors de la mise en cache. Par exemple, Vary: Accept-Encoding signifie que les versions compressées et non compressées d'une ressource doivent être mises en cache séparément.

Lorsqu'une ressource est mise en cache, elle peut être validée avant d'être réutilisée afin de s'assurer qu'elle est toujours à jour. Il existe deux mécanismes principaux pour cela :

1. **Validation forte avec ETag** : Le client envoie une requête avec l'en-tête If-None-Match contenant l'ETag de la ressource mise en cache. Si le serveur répond avec un statut 304 Not Modified, cela signifie que la ressource n'a pas changé et peut être réutilisée à partir du cache.
2. **Validation faible avec Last-Modified** : Le client peut envoyer une requête avec l'en-tête If-Modified-Since contenant la date de la dernière modification de la ressource. Le serveur renverra 304 Not Modified si la ressource n'a pas changé depuis cette date.

Le caching peut être configuré selon différents modèles en fonction des besoins :

1. **Cache-First** : Le client consulte d'abord le cache local pour une ressource et ne fait une requête au serveur que si la ressource n'est pas présente dans le cache.
2. **Network-First** : Le client fait une requête au serveur et n'utilise le cache que si la requête échoue ou si la réponse n'est pas disponible.
3. **Cache then Network** : La ressource est servie à partir du cache immédiatement, mais une requête réseau est effectuée en arrière-plan pour mettre à jour la ressource en cache.

Lorsque les ressources en cache deviennent obsolètes (selon Cache-Control ou Expires), le cache doit les revalider avec le serveur pour s'assurer qu'elles sont toujours valides. Si elles sont périmées, une nouvelle version est téléchargée, sinon le cache peut continuer à servir la ressource existante.

Les **cookies** et les **sessions** sont des mécanismes essentiels dans les applications web pour gérer l'état et suivre les utilisateurs à travers leurs interactions sur un site web. Ces deux concepts, bien que souvent utilisés ensemble, fonctionnent différemment et répondent à des besoins distincts.

Les **cookies** sont de petits fichiers texte stockés par le navigateur sur l'appareil de l'utilisateur. Ils permettent aux sites web de stocker des informations spécifiques concernant un utilisateur et de les retrouver lors des visites suivantes. Les cookies sont couramment utilisés pour :

1. **Gestion de sessions** : Identifier un utilisateur pendant qu'il est connecté à un site web. Par exemple, un cookie peut contenir un identifiant de session qui permet au serveur de reconnaître l'utilisateur lorsqu'il passe d'une page à l'autre, sans qu'il ait à se reconnecter à chaque fois.
2. **Personnalisation** : Enregistrer les préférences de l'utilisateur (langue, thème, etc.), afin de personnaliser l'expérience lors des futures visites.
3. **Suivi des utilisateurs** : Les cookies peuvent être utilisés à des fins d'analyse ou de publicité pour suivre les utilisateurs à travers plusieurs sessions ou sites web (tracking). Par exemple, des

entreprises peuvent utiliser des cookies pour collecter des informations sur les habitudes de navigation d'un utilisateur.

Il existe trois types de cookies :

1. **Cookies persistants** : Ceux-ci sont stockés sur le disque dur de l'utilisateur et ont une date d'expiration définie par l'en-tête Expires ou Max-Age. Ils restent sur le navigateur même après la fermeture du navigateur, jusqu'à leur expiration ou leur suppression.
2. **Cookies de session** : Ceux-ci ne sont stockés que pendant la durée de la session de navigation et sont supprimés lorsque l'utilisateur ferme son navigateur. Ils sont principalement utilisés pour garder les utilisateurs connectés à un site web ou pour maintenir le panier d'achat dans une boutique en ligne.
3. **Cookies tiers** : Ceux qui sont définis par des domaines autres que celui de la page que l'utilisateur visite. Par exemple, des cookies de publicité sont souvent placés par des services tiers pour suivre les utilisateurs à travers différents sites.

Une **session** est un mécanisme côté serveur permettant de stocker des informations spécifiques à l'utilisateur pendant une interaction donnée. Contrairement aux cookies, les sessions ne stockent pas directement de données sur l'appareil de l'utilisateur, mais seulement un identifiant unique (souvent dans un cookie de session) qui permet au serveur d'identifier la session correspondante.

Une session fonctionne de la manière suivante :

1. **Création de la session** : Lorsqu'un utilisateur se connecte à un site web ou effectue une action nécessitant un suivi (comme ajouter des articles à un panier), le serveur crée une session. Il associe cette session à un identifiant unique (Session ID).
2. **Stockage des données** : Les données de session sont stockées côté serveur. Cela peut inclure des informations telles que l'état de connexion de l'utilisateur, son panier d'achat, ou d'autres préférences spécifiques.
3. **Identifiant de session** : L'identifiant de session est généralement stocké dans un cookie de session ou parfois dans l'URL (bien que cette dernière méthode soit moins sécurisée). À chaque requête, le navigateur renvoie cet identifiant, ce qui permet au serveur de retrouver la session correspondante.
4. **Fin de session** : Les sessions peuvent expirer après une certaine période d'inactivité, ou elles peuvent être explicitement fermées lorsque l'utilisateur se déconnecte du site. Une fois la session terminée, les données associées sont généralement supprimées du serveur.

Les **requêtes HTTP Range** permettent de demander seulement une partie spécifique d'une ressource au lieu de la télécharger entièrement. Cela est particulièrement utile pour les fichiers volumineux (comme des vidéos, des fichiers audios, ou des documents) ou dans des situations où la reprise d'un téléchargement interrompu est nécessaire.

Une requête HTTP standard télécharge l'intégralité d'une ressource. Cependant, dans certaines situations, il peut être avantageux de ne récupérer qu'une partie du contenu. Les requêtes HTTP Range permettent de demander une ou plusieurs sous-parties spécifiques d'une ressource. Cela est rendu possible via l'en-tête Range, qui permet au client de spécifier une ou plusieurs plages d'octets (bytes) dans une ressource à récupérer.

1. **Client** : Lorsqu'un client (comme un navigateur ou une application de téléchargement) souhaite récupérer une portion d'un fichier, il envoie une requête HTTP avec un en-tête Range qui spécifie les octets ou plages d'octets qu'il veut télécharger (ex : Range: bytes=0-499 (demande les 500 premiers octets de la ressource ; Range: bytes=0-499, 1000-1499 demande les octets 0 à 499 et 1000 à 1499)).
2. **Serveur** : Si le serveur supporte les requêtes Range, il renvoie une réponse avec le code d'état 206 Partial Content, indiquant que seulement une portion de la ressource a été envoyée. Le serveur inclut également un en-tête Content-Range spécifiant quelle portion du fichier a été envoyée (ex : Content-Range: bytes 0-499/1234 indique que les octets de 0 à 499 d'une ressource totale de 1234 octets ont été envoyés).
3. Client : Le client reçoit la portion demandée du fichier et peut l'afficher ou l'utiliser. Il peut ensuite faire d'autres requêtes Range pour récupérer d'autres parties de la ressource, si nécessaire.

On utilise les requêtes HTTP Range dans les cas suivants :

1. **Reprise de téléchargement interrompu** : L'un des cas d'utilisation les plus courants des requêtes Range est la reprise de téléchargements interrompus. Si un téléchargement est interrompu (en

raison d'une coupure de connexion ou d'une interruption de service), un client peut demander seulement la partie restante du fichier en utilisant une requête Range (ex : Si un fichier de 10 Mo a été téléchargé à 70 %, une nouvelle requête HTTP avec Range : bytes=7000000 - demandera les octets à partir de 7 Mo jusqu'à la fin du fichier).

2. **Streaming de vidéos ou d'audio :** Les requêtes Range sont fréquemment utilisées pour le streaming multimédia. Lorsque vous regardez une vidéo ou écoutez de la musique en ligne, le lecteur multimédia demande des portions spécifiques du fichier en fonction du moment où vous êtes dans la lecture (par exemple, vous pouvez avancer ou reculer dans la vidéo). Le client demande une plage d'octets qui correspond à la partie de la vidéo ou de l'audio à lire, et le serveur répond avec cette partie spécifique, permettant ainsi la lecture sans télécharger l'intégralité du fichier.
3. **Visualisation progressive des fichiers PDF ou images :** Certains navigateurs et applications de visualisation de documents utilisent des requêtes Range pour permettre un rendu progressif des documents. Par exemple, si vous ouvrez un fichier PDF volumineux dans un navigateur, seule une partie du fichier est d'abord téléchargée, puis d'autres parties sont récupérées au fur et à mesure que vous faites défiler ou zoomer dans le document.

L'en-tête Range est utilisé dans la requête HTTP pour demander une ou plusieurs parties spécifiques d'une ressource. Cet en-tête prend plusieurs formes, notamment :

Range: bytes=0-499 : Demande les 500 premiers octets.

Range: bytes=500-999 : Demande les octets de 500 à 999.

Range: bytes=-500 : Demande les 500 derniers octets.

Range: bytes=500- : Demande tous les octets à partir du 500e jusqu'à la fin du fichier.

Range: bytes=0-499, 1000-1499 : Demande plusieurs plages non contigües dans la même requête.

L'en-tête Content-Range est utilisé dans la réponse HTTP pour indiquer la portion de la ressource qui a été envoyée, ainsi que la taille totale de la ressource (ex : Content-Range: bytes 0-499/1234 : Indique que les octets 0 à 499 ont été envoyés, et que la taille totale de la ressource est de 1234 octets). Si le serveur ne peut pas satisfaire la requête Range, il renverra une réponse avec un code d'état 416 (Requested Range Not Satisfiable) et inclura un en-tête Content-Range indiquant la taille totale de la ressource disponible, comme Content-Range: bytes */1234.

Tous les serveurs web ne supportent pas les requêtes Range. Cela dépend de la configuration du serveur. Les serveurs modernes, comme Apache et Nginx, prennent généralement en charge cette fonctionnalité. Si un serveur ne supporte pas les requêtes Range, il renverra une réponse standard avec tout le contenu de la ressource, même si un en-tête Range a été inclus dans la requête.

Tous les clients HTTP peuvent faire des requêtes Range, mais certains cas d'usage sont plus courants dans les navigateurs web modernes, les gestionnaires de téléchargements, les applications de streaming vidéo et les applications de visualisation de documents.

Les **redirections HTTP** sont un mécanisme qui permet de rediriger les utilisateurs d'une URL à une autre, facilitant ainsi la gestion des ressources et améliorant l'expérience utilisateur. Ces redirections peuvent être déclenchées côté serveur ou côté client (avec JavaScript). Elles sont largement utilisées dans des situations telles que la migration d'un site web, la suppression de pages, ou encore la gestion des erreurs 404.

Les redirections HTTP sont représentées par des codes de statut 3xx, chacun ayant une signification différente :

1. **301 – Redirection permanente** : Le code de statut 301 indique que la ressource a été déplacée de façon permanente vers une nouvelle URL. C'est la méthode recommandée pour les changements d'URL définitifs, car elle informe les moteurs de recherche de transférer l'autorité et le référencement de l'ancienne page vers la nouvelle.
2. **302 – Redirection temporaire** : Le code de statut 302 indique que la ressource demandée est temporairement disponible à une autre URL. Cela suggère que l'URL d'origine pourrait être restaurée à l'avenir. Les moteurs de recherche ne transfèrent pas le SEO d'une page 302 vers la nouvelle URL.
3. **303 – See Other** : La redirection 303 est utilisée après une requête POST pour rediriger le client vers une autre URL où la ressource peut être récupérée via une requête GET. Cela évite le rechargement et la resoumission accidentelle de formulaires.

4. **307 – Redirection temporaire** (préserve la méthode HTTP) : La redirection 307 est similaire à la 302, mais garantit que la méthode HTTP (GET, POST, etc.) ne change pas entre les requêtes.
5. **308 – Redirection permanente** (préserve la méthode HTTP) : La redirection 308 est équivalente à la 301, sauf qu'elle conserve la méthode HTTP originale. Si la requête initiale était un POST, la requête redirigée serait également un POST.

Bien que ce soit généralement le rôle du serveur de rediriger les requêtes, les redirections peuvent aussi être effectuées côté client, c'est-à-dire directement via le navigateur de l'utilisateur, en utilisant JavaScript. C'est une méthode plus dynamique et interactive, permettant des redirections basées sur des conditions ou des événements particuliers (comme la soumission d'un formulaire, la validation de données, etc.). Voici plusieurs manières de réaliser une redirection en JavaScript :

1. Avec `window.location.href` : C'est la plus couramment utilisée pour effectuer des redirections en JavaScript. Elle redirige immédiatement l'utilisateur vers la nouvelle URL tout en conservant la page actuelle dans l'historique du navigateur (ex : `window.location.href = "https://www.exemple.com";`). Dans cet exemple, le navigateur redirige l'utilisateur vers `https://www.example.com`. L'utilisateur pourra revenir à la page d'origine en utilisant le bouton "retour" du navigateur.
2. Avec `window.location.replace` : Elle redirige l'utilisateur vers une nouvelle URL, mais sans ajouter la page actuelle à l'historique du navigateur. Cela empêche l'utilisateur de revenir en arrière en utilisant le bouton "retour" (ex : `window.location.replace = "https://www.exemple.com";`). Cette méthode est souvent utilisée lorsqu'une redirection permanente ou critique est nécessaire, comme après une connexion réussie ou après la soumission d'un formulaire pour éviter les doublons.
3. Avec `window.location.assign` : La méthode `window.location.assign` fonctionne de la même manière que `window.location.href`, mais elle peut être plus explicite à utiliser (ex : `window.location.assign = "https://www.exemple.com";`).

HTTP dispose de 6 méthodes standard pour indiquer l'action à effectuer lors des requêtes :

GET : Récupère une ressource du serveur.

POST : Envoie des données au serveur (formulaire, fichiers).

PUT : Remplace une ressource sur le serveur.

DELETE : Supprime une ressource.

HEAD : Similaire à GET, mais ne renvoie que les en-têtes.

OPTIONS : Renvoie les méthodes HTTP disponibles pour une ressource.

Enfin, Les **codes d'état** permettent d'indiquer le résultat d'une requête. Ils sont organisés en 5 catégories :

1xx (Informationnel) : La requête est en cours de traitement.

2xx (Succès) : La requête a réussi. Ex : 200 OK.

3xx (Redirection) : Il est nécessaire d'effectuer une autre action. Ex : 301 Moved Permanently.

4xx (Erreur client) : La requête est mal formulée. Ex : 404 Not Found.

5xx (Erreur serveur) : Le serveur a rencontré une erreur. Ex : 500 Internal Server Error.

L'API FETCH

L'API Fetch est une interface moderne et puissante qui permet d'effectuer des requêtes HTTP à partir de JavaScript de manière asynchrone. Elle remplace l'ancienne méthode `XMLHttpRequest()` et offre une syntaxe plus simple et intuitive basée sur les promesses. Cette API est souvent utilisée pour interagir avec des serveurs ou des **APIs REST** afin de récupérer ou envoyer des données.

L'API Fetch est généralement utilisée pour effectuer des opérations comme récupérer des données d'une API, envoyer des informations au serveur ou gérer des fichiers. La méthode de base `fetch()` prend en paramètre une URL et un objet facultatif, et retourne une promesse qui résout la réponse (ou l'erreur) de la requête. Voici la structure générale pour effectuer une requête HTTP avec Fetch :

```
fetch(url, options)
  .then(reponse => {
    //Traiter la réponse
  })
  .catch(erreur => {
    //Gérer les erreurs réseau
  })
```

```
});
```

Par défaut, `fetch()` effectue une requête HTTP de type GET. Toutefois, il est possible de spécifier d'autres méthodes (comme POST, PUT, DELETE, etc.) via le paramètre **options** (ex :

```
fetch('https://api.exemple.com/data')
  .then(reponse => reponse.json()) //Récupérer et traiter la réponse
  .then(data => console.log(data)) //Afficher les données reçues
  .catch(erreur => console.error('Erreur:', erreur)); //Gérer les erreurs.
```

Le deuxième argument de la fonction `fetch()` est un objet d'options, et il peut contenir plusieurs propriétés qui permettent de configurer la requête. Voici quelques-unes des propriétés que vous pouvez inclure dans cet objet :

1. **method** : Spécifie la méthode HTTP à utiliser, comme GET, POST, PUT, DELETE, etc.
(ex : `method: 'POST'`).
2. **headers** : Permet de définir les en-têtes HTTP à envoyer avec la requête. C'est généralement un objet avec des paires clé-valeur (ex :

```
headers: {
  'Content-Type': 'application/json',
  'Authorization': 'Bearer token'
}.
```
3. **body** : Contient le corps de la requête, souvent utilisé avec les méthodes POST ou PUT. Il peut s'agir d'une chaîne de caractères, d'un objet `FormData`, ou d'autres types de données (ex :
`body: JSON.stringify({key: 'value'})`).
4. **mode** : Définit le mode de la requête, comme cors, no-cors, ou same-origin.
(ex : `mode: 'cors'`).
5. **credentials** : Indique si les cookies doivent être envoyés avec la requête. Les valeurs possibles sont same-origin, include, ou omit.
(ex : `credentials: 'include'`).
6. **cache** : Contrôle la politique de mise en cache. Les valeurs possibles sont default, no-store, reload, no-cache, force-cache, ou only-if-cached.
(ex : `cache: 'no-cache'`).
7. **redirect** : Spécifie comment les redirections doivent être traitées. Les valeurs possibles sont follow, manual, ou error.
(ex : `redirect: 'follow'`).
8. **referrer** : Définit la valeur de l'en-tête Referer. Cela peut être une URL ou une chaîne vide pour supprimer l'en-tête.
(ex : `referrer: ''`).
9. **referrerPolicy** : Définit la politique de la referrer. Les valeurs possibles incluent no-referrer, no-referrer-when-downgrade, origin, origin-when-cross-origin, same-origin, ou strict-origin-when-cross-origin.

Les requêtes **GET** sont les plus couramment utilisées pour récupérer des données d'un serveur. Elles sont utilisées sans fournir un corps de requête, et les paramètres peuvent être ajoutés directement à l'URL (ex :

```
fetch('https://api.example.com/users?id=${userId}')
  .then(reponse => reponse.json())
  .then(donnée => {
    console.log('Utilisateurs:', donnée);
  })
  .catch(erreur => console.error('Erreur:', erreur));
```

Les requêtes **POST** permettent d'envoyer des données au serveur. Elles sont souvent utilisées pour soumettre des formulaires, créer de nouvelles ressources, ou envoyer des données complexes comme des objets JSON (ex :

```
fetch('https://api.exemple.com/users', {
  method: 'POST', //Spécifie la méthode POST
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
```

```

nom: 'Chris Koyo',
email: 'chk@example.com'
}) //Le corps de la requête est sérialisé en JSON
).then(reponse => {
  if (!reponse.ok){
    throw new Error(`Erreur HTTP: ${reponse.status}`);
  }
  return reponse.json();
})
.then(donnée => console.log('Utilisateur créé:', donnée))
.catch(erreur => console.error('Erreur:', erreur)); Dans cet exemple, nous envoyons un
objet JavaScript sous forme de JSON au serveur en utilisant l'en-tête Content-Type: application/json.

```

Les requêtes **PUT** et **PATCH** sont utilisées pour mettre à jour une ressource existante sur le serveur. La différence principale entre les deux est que PUT remplace généralement entièrement la ressource, tandis que PATCH effectue une mise à jour partielle (ex :

```

fetch('https://api.example.com/users/1', {
  method: 'PUT', //Spécifie la méthode PUT
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    nom: 'Chris Koyo',
    email: 'chk@example.com'
}) //Le corps de la requête est sérialisé en JSON
).then(reponse => {
  if (!reponse.ok){
    throw new Error(`Erreur HTTP: ${reponse.status}`);
  }
  return reponse.json();
})
.then(donnée => console.log('Utilisateur mis à jour:', donnée))
.catch(erreur => console.error('Erreur:', erreur));

```

Les requêtes **DELETE** sont utilisées pour supprimer une ressource du serveur. Elles sont souvent accompagnées d'un identifiant dans l'URL pour indiquer la ressource à supprimer (ex :

```

fetch('https://api.example.com/users', {
  method: 'DELETE', //Spécifie la méthode DELETE
}).then(reponse => {
  if (!reponse.ok){
    throw new Error(`Erreur HTTP: ${reponse.status}`);
  }
  console.log('Utilisateur supprimé avec succès');
})
.catch(erreur => console.error('Erreur:', erreur));

```

L'API Fetch ne rejette pas automatiquement les promesses pour les erreurs liées aux réponses HTTP (par exemple, un code de statut 404 ou 500). Vous devez explicitement vérifier si la réponse est correcte en utilisant la propriété `ok` de l'objet `reponse`.

Une autre source d'erreurs peut être une perte de connexion réseau ou des problèmes de serveur. Ces erreurs doivent également être capturées via un `catch()`.

Fetch simplifie l'envoi de formulaires sans avoir besoin de recharger la page. Voici un exemple d'envoi de données de formulaire :

```

<!DOCTYPE html>
<html lang="fr">
<head>
<meta charset="UTF-8">

```

```

<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>monFormulaireHTML</title>
</head>
<body>
<form id="userForm">
  <input type="text" name="name" placeholder="Nom" />
  <input type="email" name="email" placeholder="Email" />
  <button type="submit">Envoyer</button>
</form>

<script>
  document.getElementById('userForm').addEventListener('submit', function(event) {
    event.preventDefault(); //Empêche le rechargement de la page
    const formData = new FormData(event.target); //Récupère les données du formulaire

    fetch('https://api.exemple.com/users', {
      method: 'POST',
      body: formData //Envoie les données du formulaire
    })
    .then(reponse => reponse.json())
    .then(donnée => console.log('Formulaire soumis avec succès:', donnée))
    .catch(erreur => console.error('Erreur:', erreur));
  });
</script>
</body>
</html>

```

Le corps d'une requête ou d'une réponse peut contenir différentes données : JSON, XML, texte brut, fichiers binaires, etc. L'objet `reponse` représente la réponse HTTP d'une requête Fetch. Il contient des méthodes comme `json()`, `text()`, `blob()`, etc., qui permettent de lire le contenu de la réponse.

La promesse renvoyée par `fetch()` contient un objet `reponse`, qui a plusieurs propriétés importantes pour le traitement du corps de la réponse :

1. `ok` : Un booléen qui indique si la requête s'est bien passée (valeur `true` si le statut HTTP est compris entre 200 et 299).
2. `status` : Le code de statut HTTP de la réponse (par exemple, 200 pour OK, 404 pour Not Found).
3. `statusText` : Le message correspondant au statut HTTP (par exemple, 'OK' ou 'Not Found').
4. `headers` : Les en-têtes de la réponse, disponibles via un objet `Headers`.
5. `body` : Le flux de données de la réponse, qui peut être lu de plusieurs façons (selon le format des données).

L'objet `reponse` fournit aussi plusieurs méthodes pour traiter le corps de la réponse, en fonction du type de données attendu. Voici les méthodes les plus courantes :

1. `reponse.json()` : Cette méthode convertit le corps de la réponse en un objet JavaScript à partir de données au format JSON. C'est l'une des méthodes les plus utilisées, car les API modernes renvoient souvent des données en JSON (ex :

```

fetch('https://api.exemple.com/data')
  .then(reponse => {
    if (!reponse.ok){
      throw new Error('Erreur réseau');
    }
    return reponse.json();
  })
  .then(données => {
    console.log('Données JSON:', données);
  })
  .catch(erreur => {
    console.error('Erreur:', erreur);
  });

```

2. `reponse.text()` : Cette méthode traite le corps de la réponse comme une chaîne de texte brute. Cela est utile lorsque vous devez traiter des fichiers texte (comme des fichiers .txt ou du code HTML) (ex :


```
fetch('https://exemple.com/text')
  .then(reponse => reponse.text())
  .then(texte => {
    console.log('Texte:', texte);
});.
```
3. `reponse.blob()` : Cette méthode convertit la réponse en un objet `Blob`, qui représente des données binaires telles que des images, des fichiers PDF, ou des vidéos (ex :


```
fetch('https://exemple.com/image.jpg')
  .then(reponse => reponse.blob())
  .then(monBlob => {
    const url = URL.createObjectURL(monBlob);
    const image = document.createElement('img');
    img.src = url;
    document.body.appendChild(image);
});.
```
4. `reponse.arrayBuffer()` : Cette méthode traite le corps de la réponse sous forme d'`ArrayBuffer`, qui représente une séquence de données binaires. Cette méthode est souvent utilisée pour des cas avancés, tels que le traitement de fichiers multimédias, ou lorsqu'on souhaite manipuler des données binaires (ex :


```
fetch('https://exemple.com/data')
  .then(reponse => reponse.arrayBuffer())
  .then(buffer => {
    //Traiter le tableau d'octets
    console.log('ArrayBuffer:', buffer);
});.
```
5. `reponse.formData()` : Si la réponse est une structure encodée en multipart/form-data, vous pouvez utiliser cette méthode pour récupérer un objet `FormData` (ex :


```
fetch('https://exemple.com/form')
  .then(reponse => reponse.formData())
  .then(formDonnées => {
    for (let [clé, valeur] of formDonnées.entries()){
      console.log(`#${clé}: ${valeur}`);
    }
});.
```

Il est toujours important de vérifier si la réponse est correcte avant de traiter le corps. Utilisez `reponse.ok` pour vérifier les codes de statut.

`fetch()` ne traite pas automatiquement les erreurs réseau. Vous devez toujours prévoir un `catch()` pour intercepeter les erreurs.

L'API WEB STORAGE

L'API Web Storage en JavaScript est une interface puissante permettant aux applications web de stocker des données côté client, à l'intérieur du navigateur. Ces données peuvent être stockées de manière persistante (via `localStorage`) ou temporairement (via `sessionStorage`). Voici un guide complet qui aborde les différents aspects, fonctions, objets, et événements liés à cette API.

Les deux objets principaux fournis par l'API Web Storage sont :

1. `localStorage` : Stocke les données de manière persistante, c'est-à-dire que les données seront toujours disponibles même après la fermeture du navigateur ou un redémarrage de l'ordinateur.
2. `sessionStorage` : Stocke les données uniquement pour la durée de la session du navigateur. Ces données sont effacées dès que l'onglet ou la fenêtre contenant la page est fermée.

Ces deux objets partagent une API similaire, mais ils ont des comportements différents quant à la durée de vie des données.

Ces deux objets partagent cinq méthodes clés pour manipuler les données :

1. **setItem(*clé, valeur*)** : Cette méthode permet de stocker une donnée dans le stockage, sous la forme d'une paire clé-valeur. La clé est utilisée pour récupérer ou supprimer la valeur plus tard.
Paramètres :
 - *clé* (string) : Le nom de la clé sous laquelle la donnée sera stockée.
 - *valeur* (string) : La donnée à stocker, qui doit être une chaîne de caractères. Si vous devez stocker des objets ou des tableaux, il faut les convertir en JSON.

(ex :

```
//Stockage dans localStorage
localStorage.setItem('nomUtilisateur', 'Jean');
//Stockage dans sessionStorage
sessionStorage.setItem('sessionId', '12345');
```
2. **getItem(*clé*)** : Cette méthode est utilisée pour récupérer une donnée stockée dans `localStorage` ou `sessionStorage` en fonction de la clé. Elle retourne la valeur sous forme de chaîne de caractères, ou `null` si la clé n'existe pas.
Paramètre : *clé* (string) : Le nom de la clé associée à la donnée.
(ex :


```
//Récupérer une donnée depuis localStorage
let nomUtilisateur = localStorage.getItem('nomUtilisateur');
console.log(nomUtilisateur); //Affiche "Jean"
```



```
//Récupérer une donnée depuis sessionStorage
let sessionId = sessionStorage.getItem('sessionId');
console.log(sessionId); //Affiche "12345".
```
3. **removeItem(*clé*)** : Cette méthode permet de supprimer une donnée stockée en fonction de sa clé.
Paramètre : *clé* (string) : Le nom de la clé associée à la donnée à supprimer.
(ex :


```
//Supprimer une donnée dans localStorage
localStorage.removeItem('nomUtilisateur');
```



```
//Supprimer une donnée dans sessionStorage
sessionStorage.removeItem('sessionId');
```
4. **clear()** : Cette méthode supprime toutes les paires clé-valeur stockées dans `localStorage` ou `sessionStorage` (ex :


```
//Effacer toutes les données dans localStorage
localStorage.clear();
```



```
//Effacer toutes les données dans sessionStorage
sessionStorage.clear();
```
5. **key(index)** : Cette méthode retourne la clé à la position index spécifiée dans le stockage. Elle permet de parcourir toutes les paires clé-valeur stockées. Elle retourne la clé sous forme de chaîne de caractères.
Paramètre : *index* (number) : L'index de la clé souhaitée.
(ex :


```
//Parcourir les clés de localStorage
for (let i = 0; i < localStorage.length; i++){
    console.log(localStorage.key(i));
}
```

Les objets `localStorage` et `sessionStorage` partagent certaines caractéristiques intéressantes :

- Chaque navigateur a une limite de stockage d'environ 5 MB par domaine (cela peut varier).
- Les données stockées ne sont accessibles que par les pages du même domaine.
- Les données sont toujours stockées sous forme de chaînes de caractères. Si vous souhaitez stocker des objets, vous devez les convertir en JSON.

L'événement '`storage`' est déclenché chaque fois qu'une modification est effectuée dans le stockage d'une page web (mais uniquement si cette modification est faite depuis une autre page ou un autre onglet du même domaine). Les propriétés de l'événement '`storage`' sont :

`key` : La clé qui a été modifiée (ou `null` si `clear()` a été appelé).

`oldValue` : La valeur précédente de la clé, avant la modification.

`newValue` : La nouvelle valeur de la clé après modification.

`url` : L'URL du document qui a modifié le stockage.

(ex :

```
window.addEventListener('storage', function(e){
  console.log('Changement détecté dans le stockage :');
  console.log('Clé : ' + e.key);
  console.log('Ancienne valeur : ' + e.oldValue);
  console.log('Nouvelle valeur : ' + e.newValue);
  console.log('URL : ' + e.url);
});).
```

L'API Web Storage est utilisée dans de nombreux scénarios pour améliorer l'expérience utilisateur. Voici quelques exemples :

- Conservation des préférences utilisateur : Les paramètres comme le thème sombre, les préférences de langue ou les filtres de recherche peuvent être stockés dans `localStorage` pour que l'utilisateur les retrouve à chaque visite.
- Sessions utilisateur : Vous pouvez utiliser `sessionStorage` pour conserver les informations de session pendant la durée de la visite (par exemple, un panier d'achat).
- Gestion d'états d'authentification : `localStorage` peut stocker des jetons d'authentification (JWT tokens) pour permettre aux utilisateurs de rester connectés entre les sessions.

Les données stockées dans `localStorage` ou `sessionStorage` sont accessibles à tout script exécuté sur la page, ce qui signifie que vous ne devriez jamais stocker d'informations sensibles (comme des mots de passe) dans le Web Storage sans les chiffrer au préalable.

Avec une limite de stockage relativement faible, il est déconseillé de stocker des quantités massives de données dans le Web Storage. Pour des besoins de stockage plus importants, il est recommandé d'explorer d'autres options comme `IndexedDB`.

`localStorage` permet de synchroniser des données entre plusieurs onglets du même domaine, mais `sessionStorage` ne fonctionne que pour l'onglet actif. Si vous devez synchroniser les données de session entre plusieurs onglets, vous devrez implémenter des mécanismes spécifiques.

Les cookies en JavaScript est un mécanisme de stockage côté client qui permet aux sites web de stocker de petites quantités de données (jusqu'à 4 KB par cookie) sous forme de paires clé-valeur dans le navigateur. Contrairement à d'autres options de stockage comme `localStorage` ou `sessionStorage`, les cookies sont automatiquement envoyés avec chaque requête HTTP au serveur, ce qui les rend utiles pour la gestion de sessions ou le suivi des utilisateurs sur différentes pages d'un site web.

Chaque cookie ne peut pas dépasser environ 4 KB. Ils peuvent être configurés pour expirer après une certaine période ou être effacés à la fermeture du navigateur (cookies de session). Ils peuvent être lus et écrits aussi bien côté serveur (via les en-têtes HTTP) que côté client (via JavaScript). À chaque requête HTTP vers le serveur, tous les cookies correspondant au domaine sont automatiquement envoyés avec les en-têtes de la requête.

Les cookies sont stockés sous la forme de paires clé-valeur. Chaque cookie a des propriétés optionnelles telles que la date d'expiration, le chemin, le domaine et l'indicateur de sécurité.

Les cookies peuvent être définis directement en utilisant `document.cookie`. Cependant, il faut noter que `document.cookie` n'est pas une interface qui permet de manipuler des cookies comme un objet JavaScript normal. Chaque fois que vous définissez un cookie, vous modifiez la chaîne de cookies existante (ex : `document.cookie = "nom=Jean; expires=Fri, 31 Dec 2024 23:59:59 GMT; path=/";`). On définit le cookie avec une clé "nom" et une valeur "Jean". `expires=...` Définit une date d'expiration après laquelle le cookie sera supprimé (si non spécifié, le cookie est un cookie de session, c'est-à-dire qu'il sera supprimé

à la fermeture du navigateur). `path=/` : Spécifie que le cookie est accessible sur tout le site (toutes les URL à partir de la racine).

Tous les cookies d'un domaine sont accessibles via la propriété `document.cookie`. Cependant, cette propriété renvoie tous les cookies sous forme de chaîne de caractères, ce qui nécessite un traitement manuel pour extraire un cookie spécifique (ex :

```
let cookies = document.cookie; //Ex : "nom=Jean; age=25; sessionID=abc123"
console.log(cookies);
```

//Pour obtenir la valeur d'un cookie spécifique, il faut les traiter

```
function getCookie(nom){
    let name = nom + "=";
    let decodedCookie = decodeURIComponent(document.cookie);
    let ca = decodedCookie.split(';');
    for(let i = 0; i < ca.length; i++){
        let c = ca[i].trim();
        if (c.indexOf(name) == 0){
            return c.substring(name.length, c.length);
        }
    }
    return "";
}
```

```
let nomUtilisateur = getCookie("nom");
console.log(nomUtilisateur); // "Jean".
```

Pour mettre à jour un cookie, il suffit de le redéfinir avec la même clé, mais une nouvelle valeur.

La suppression d'un cookie s'effectue en le redéfinissant avec une date d'expiration passée (ex :

```
document.cookie = "nom=; expires=Thu, 01 Jan 1970 00:00:00 GMT; path=/";).
```

Les cookies disposent de plusieurs attributs facultatifs qui permettent de contrôler leur comportement :

- **`expires`** : Définit la date et l'heure d'expiration d'un cookie. Si cet attribut n'est pas spécifié, le cookie est supprimé à la fermeture du navigateur (ex : `document.cookie = "nom=Jean; expires=Fri, 31 Dec 2024 23:59:59 GMT";`).
- **`max-age`** : Spécifie la durée de vie du cookie en secondes. Si `max-age=0`, le cookie est supprimé immédiatement (ex : `document.cookie = "nom=Jean; max-age=3600"; //1 heure`).
- **`path`** : Définit le chemin URL où le cookie est accessible. Par exemple, si le chemin est `/admin`, le cookie sera disponible uniquement pour les pages de ce répertoire (ex : `document.cookie = "nom=Jean; path=/admin";`).
- **`domain`** : Spécifie le domaine pour lequel le cookie est valide. Cela permet de partager des cookies entre sous-domaines (ex : `document.cookie = "nom=Jean; domain=example.com";`).
- **`secure`** : Si cet attribut est défini, le cookie ne sera transmis que sur des connexions HTTPS sécurisées (ex : `document.cookie = "nom=Jean; secure";`).
- **`SameSite`** : Cette option contrôle si le cookie est envoyé lors des requêtes entre sites web (cross-site requests), un mécanisme utilisé pour protéger contre les attaques CSRF (Cross-Site Request Forgery).
- **`Strict`** : Le cookie n'est envoyé que lors de la navigation directe sur le site.
- **`Lax`** : Le cookie est envoyé lors des requêtes GET intersites (ex : lorsque l'utilisateur clique sur un lien menant au site).
- **`None`** : Le cookie est envoyé dans toutes les requêtes, y compris intersites (nécessite le flag `secure` pour les connexions HTTPS).

Les cookies sont souvent utilisés pour stocker des identifiants de session. Lorsqu'un utilisateur se connecte, un identifiant unique est généré côté serveur et envoyé au client sous forme de cookie (ex :

```
document.cookie = "sessionID=abc123; expires=Fri, 31 Dec 2024 23:59:59 GMT; path=/";).
```

Ensuite, à chaque requête HTTP suivante, ce cookie est renvoyé au serveur, permettant au serveur d'identifier l'utilisateur et de maintenir sa session active.

Les cookies sont aussi souvent utilisés pour suivre les activités des utilisateurs, notamment dans les publicités en ligne.

Les cookies peuvent être utilisés pour enregistrer des préférences utilisateur comme le thème choisi ou la langue préférée (ex :

```
document.cookie = "theme=sombre; expires=Fri, 31 Dec 2024 23:59:59 GMT; path=/";
```

Voici une manière pratique de gérer les cookies en encapsulant les opérations courantes dans des fonctions :

```
//Fonction pour créer un cookie
function setCookie(nom, valeur, nbrJours){
    let date = new Date();
    date.setTime(date.getTime() + (nbrJours * 24 * 60 * 60 * 1000));
    let expires = "expires=" + date.toUTCString();
    document.cookie = nom + "=" + valeur + "; " + expires + "; path=/";
}

//Fonction pour lire un cookie
function getCookie(nom){
    let name = nom + "=";
    let decodedCookie = decodeURIComponent(document.cookie);
    let ca = decodedCookie.split(';');
    for(let i = 0; i < ca.length; i++){
        let c = ca[i].trim();
        if (c.indexOf(name) == 0){
            return c.substring(name.length, c.length);
        }
    }
    return "";
}

//Fonction pour supprimer un cookie
function deleteCookie(nom){
    document.cookie = nom + "=; expires=Thu, 01 Jan 1970 00:00:00 GMT"
}
```

Maintenant, libre à vous de découvrir les autres API Web ! Pourquoi pas d'ailleurs aller plus loin et apprendre des Framework de Frontend comme React, Angular, etc.