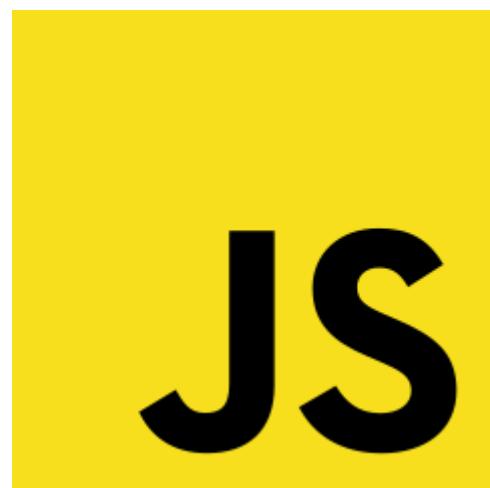
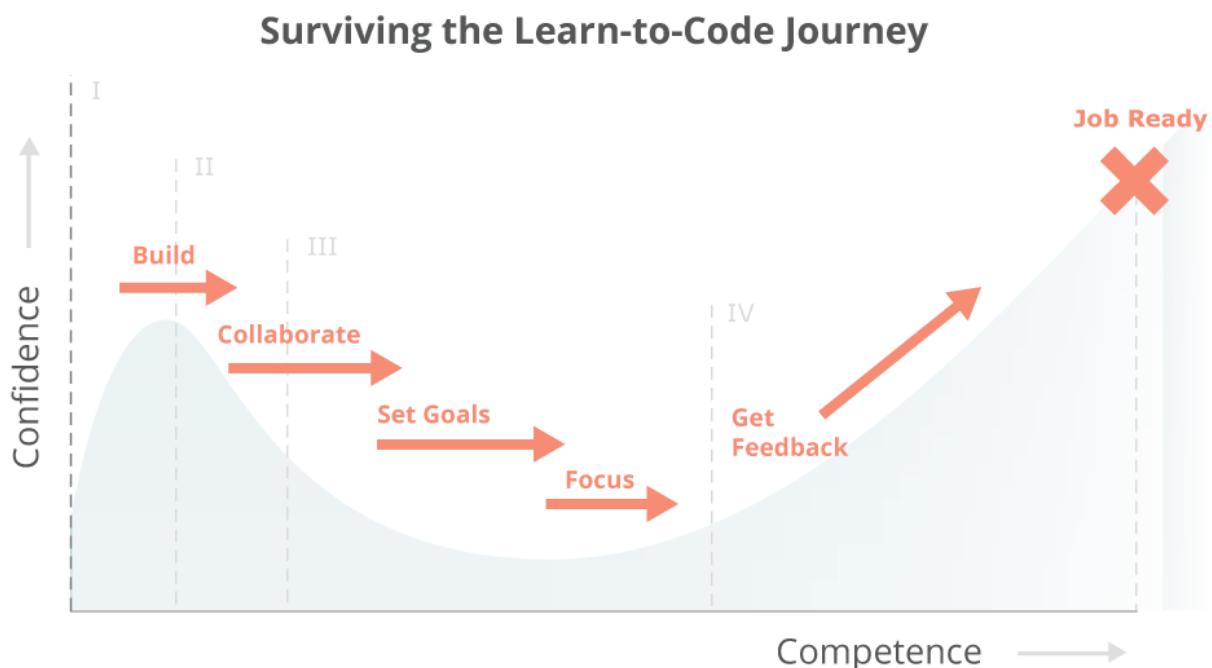


JAVASCRIPT

JAVASCRIPT



1. Apprendre à coder



2. Le langage JavaScript

2.1. Typage faible et dynamique

- JavaScript est un langage dont le typage est *faible* et *dynamique*.
- *faible* = il n'est pas nécessaire de déclarer le type d'une variable avant de l'utiliser.
- *dynamique* = le type de la variable sera automatiquement déterminé lorsque le programme sera exécuté => la même variable pourra avoir différents types au cours de son existence.

3. Commentaires

La syntaxe utilisée pour les **commentaires** est la même que celle utilisée par le C++ et d'autres langages :

```
// un commentaire sur une ligne

/* un commentaire plus
   long sur plusieurs lignes
*/

/* Par contre on ne peut pas /* imbriquer des commentaires */ SyntaxError */
```

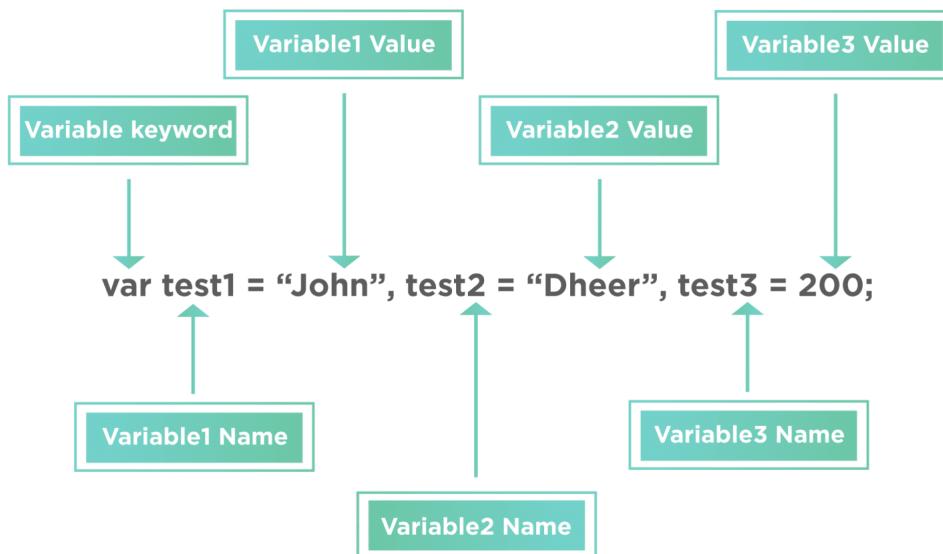
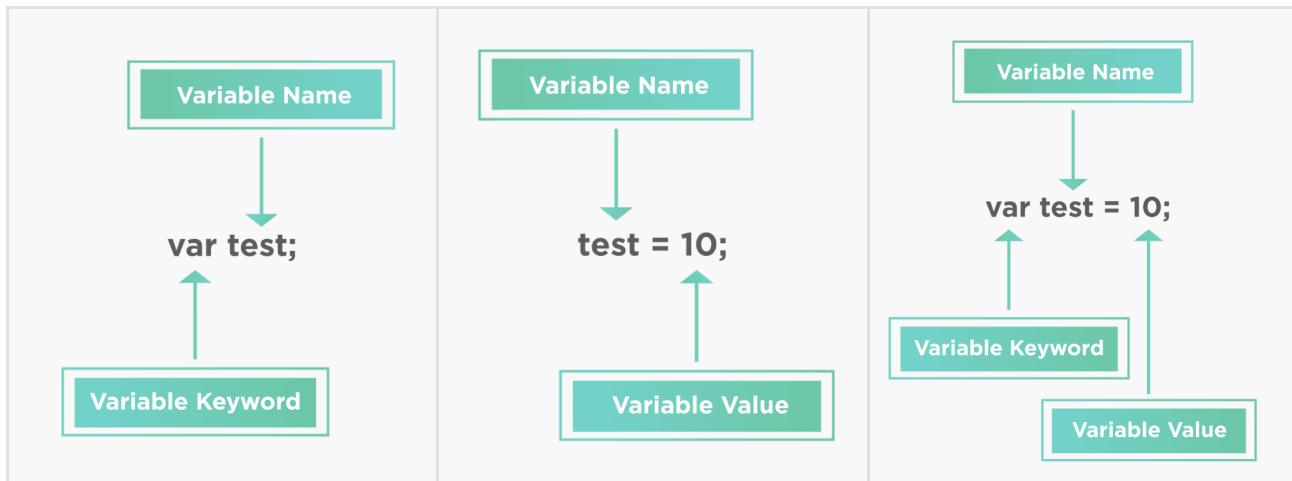
4. Variables



variable.js

Une variable est une zone de stockage contentant des données que vous pouvez utiliser plus tard dans un programme.

4.1. Anatomie d'une variable



4.2. Déclaration, Initialisation, Affectation

En JS, les **déclarations** retourneront toujours `"undefined"` et les **affectations** de variables retourneront une valeur.

Variable Lifecycle in JS

Declaration

`var a`

Initialisation

`a = undefined`

Assignment

`a = 10`

Declare it

Name it

Initialize it

`var message = "Hello"`

4.3. var

L'instruction `var` permet de déclarer une variable et éventuellement d'initialiser sa valeur.

La portée d'une variable déclarée avec `var` est le **contexte d'exécution** courant, c'est-à-dire : la fonction qui contient la déclaration ou le contexte global si la variable est déclarée en dehors de toute fonction.

Si on affecte une valeur à une variable qui n'a pas été déclarée (variable déclarée sans le mot `var` ni aucun autre mot), cela devient une **variable globale** (une propriété de l'objet global) lorsque l'affectation est exécutée.

4.4. let

L'instruction `let` permet de déclarer une variable dont la **portée est limitée à celle du bloc courant** (le bloc où la variable a été déclarée), éventuellement en initialisant sa valeur.

4.5. const

La déclaration `const` permet de créer une constante nommée accessible uniquement en lecture. Cela ne signifie pas que la valeur contenue est immuable, uniquement que l'identifiant ne peut pas être réaffecté.

Autrement dit la valeur d'une constante ne peut pas être modifiée par des réaffectations ultérieures. Une constante ne peut pas être déclarée à nouveau.

VAR, LET & CONST

The image shows a grid of six cards, each containing a snippet of JavaScript code and a circular icon indicating its validity:

- Top Left:** `var x = 65;` `var x = 5;` ✓ (green checkmark)
- Top Right:** `const x = 65;` ✓ (green checkmark)
- Middle Left:** `let x = 54;` `let x = 65;` ✗ (red X)
- Middle Right:** `const x = 65;` `x = 55;` ✗ (red X)
- Bottom Left:** `let x = 54;` `x = 65;` ✓ (green checkmark)
- Bottom Right:** `const x;` `const x = 65;` `const x = 86;` ✗ (red X)

4.6. variable déclarée / variable non déclarée

Les variables déclarées sont contraintes dans le contexte d'exécution dans lequel elles sont déclarées et sont **TOUJOURS** globales.

```
function fO {
  a = 1; // Lève une exception ReferenceError en mode strict
  var b = 2;
}
fO;
console.log(a); // Affiche "1" dans la console
console.log(b); // Lève une exception ReferenceError car b n'est pas définie en dehors de f
```

4.7. Cycle de vie d'une variable

Lorsque le moteur fonctionne avec des variables, leur cycle de vie comprend les phases suivantes :

- **La phase de déclaration** consiste à enregistrer une variable dans la portée.
- **La phase d'initialisation** consiste à allouer de la mémoire et à créer une liaison pour la variable dans l'étendue. À cette étape, la variable est automatiquement initialisée avec `undefined`
- **La phase d'affectation** consiste à attribuer une valeur à la variable initialisée.

exemple :

```
var a = 4;
```

1. **Déclaration** : la variable est enregistrée et ajoutée à la chaîne de portée globale.
2. **Initialisation** : la variable `a` s'initialise à `undefined`.
3. **Assignation** : la valeur `4` est affectée à la variable `a`.

4.8. Hoisting

Les déclarations de variables (et les déclarations en général) sont traitées avant que n'importe quel autre code soit exécuté.

Ainsi, déclarer une variable n'importe où dans le code équivaut à la déclarer au début de son contexte d'exécution. Cela signifie qu'une variable peut également apparaître dans le code avant d'avoir été déclarée.

Ce comportement est appelé « **remontée** » (*hoisting* en anglais) car la déclaration de la variable est « remontée » au début de la fonction courante ou du contexte global.

```
bonjour = 2
var bonjour;
// est implicitement traité comme :
var bonjour;
bonjour = 2;
```

Étant donné ce comportement, il est recommandé de toujours déclarer les variables au début de leurs portées (le début du code global ou le début du corps de la fonction) afin de mieux (sa)voir quelles variables font partie de la fonction et lesquelles proviennent de la chaîne de portées.

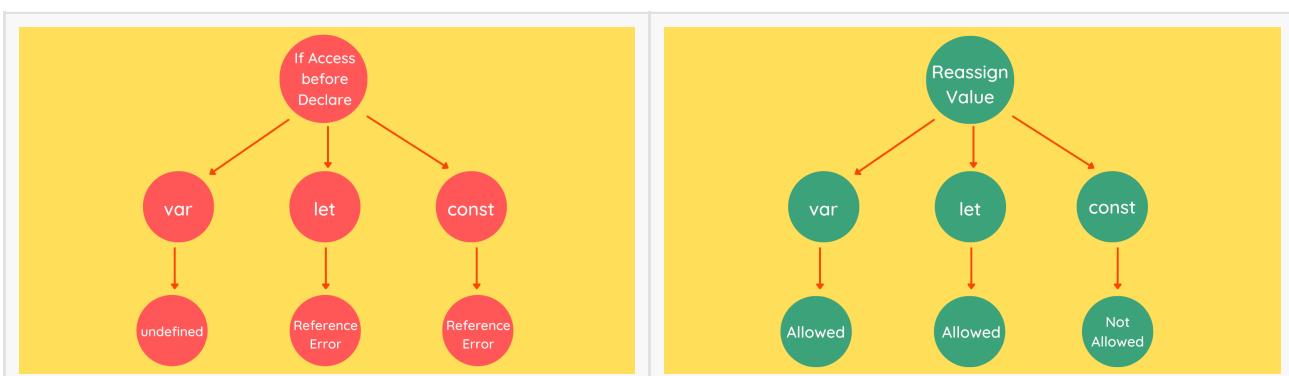
Il est important de noter que la remontée des variables affecte uniquement la déclaration et pas l'initialisation de la valeur. La valeur sera affectée lorsque le moteur accèdera à l'instruction d'affectation.

exemple :

```
console.log(pizza); // undefined
var pizza = 'Margherita';
console.log(pizza); // Margherita
// Correspond en fait à :
var pizza;
console.log(pizza); // undefined
pizza = 'Margherita';
console.log(pizza); // Margherita
```

4.9. Portée d'une variable

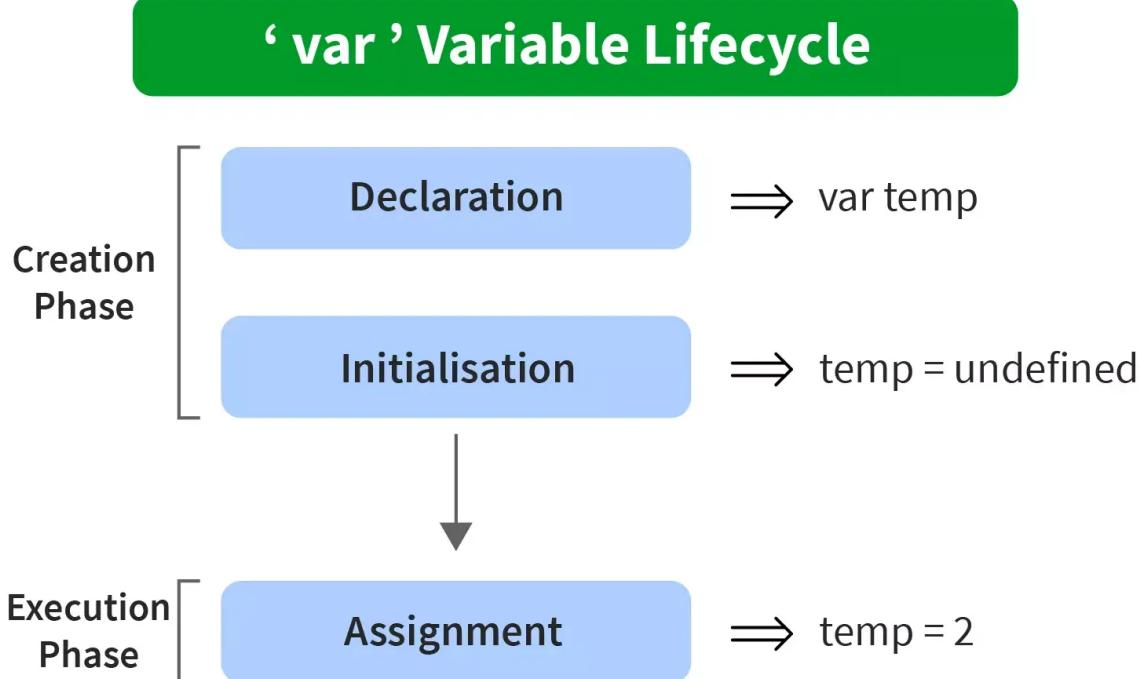
Les variables déclarées avec **const** ou **let** ne sont pas accessibles en dehors de la portée dans laquelle elles sont déclarées contrairement à **var** accessible en dehors de la portée d'un bloc, mais pas de la portée d'une fonction.



Il faut éviter de ne pas déclarer une variable car cela peut provoquer des résultats inattendus. Il est donc fortement recommandé de toujours déclarer les variables, qu'elles soient dans une fonction ou dans la portée globale.

Cycle de vie de `var`

```
console.log(temp) // declaration and initialization -> undefined
var temp = 2;    // assignment
```



Lors de la **phase de création**, la déclaration de la variable et l'initialisation avec `undefined` se produisent.

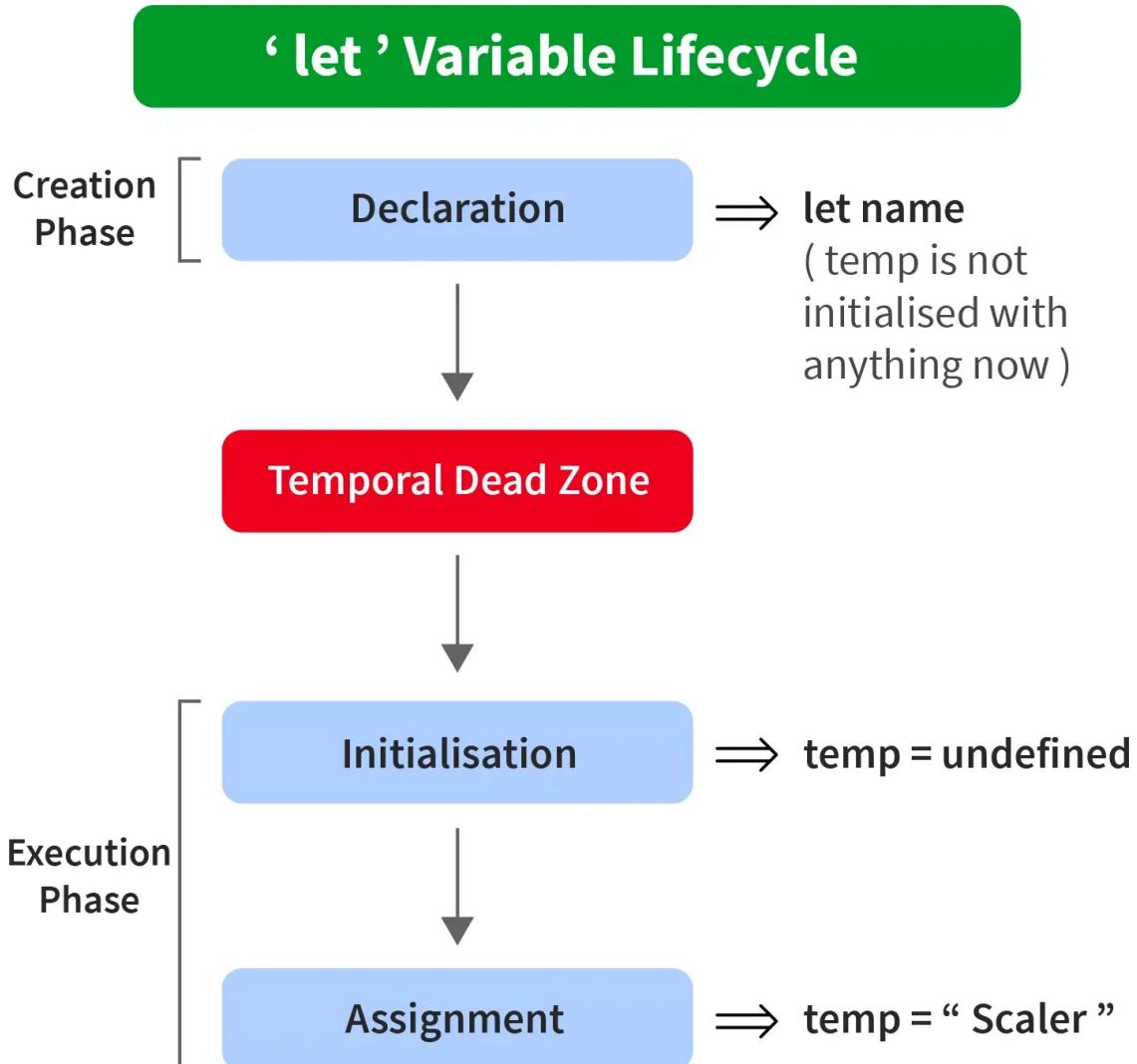
Pendant la **phase d'exécution**, lorsque le code est exécuté ligne par ligne, la variable `temp` est hissée, donc `undefined` est affiché sur la console. Ensuite, la valeur `2` est affectée à la variable.

4.10. Zone de mort temporaire (Temporal Dead Zone)

Cycle de vie de `let`

```
console.log(temp)
let temp= "scaler"
// Uncaught ReferenceError: Cannot access 'temp' before initialization
```

Nous avons obtenu une erreur que nous n'avons pas vue dans les exemples précédents. La raison en est que var est à la fois déclaré et initialisé lors du levage, mais let n'est pas initialisé, il est seulement déclaré. La variable est dans une zone morte temporelle (**TDZ**).

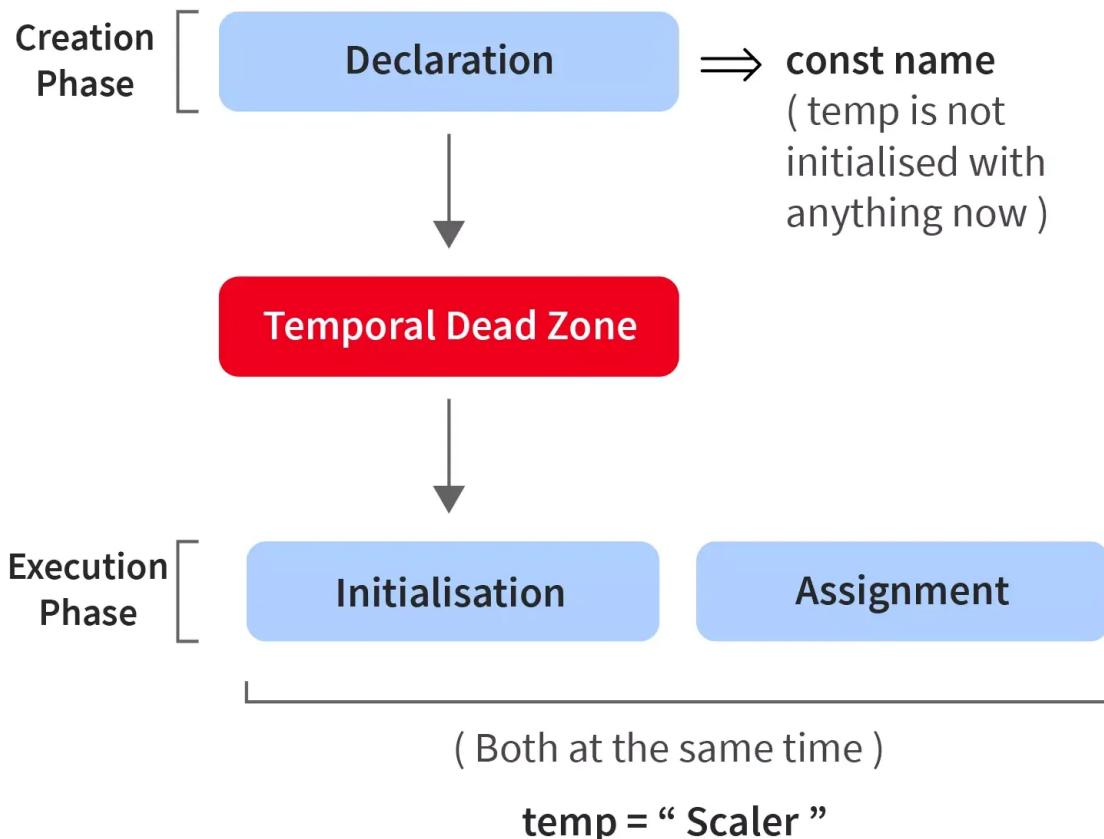


Cycle de vie de `const`

```

console.log(temp)
const temp = "Scaler"
// Uncaught ReferenceError: Cannot access 'temp' before initialization
  
```

‘`const`’ Variable Lifecycle



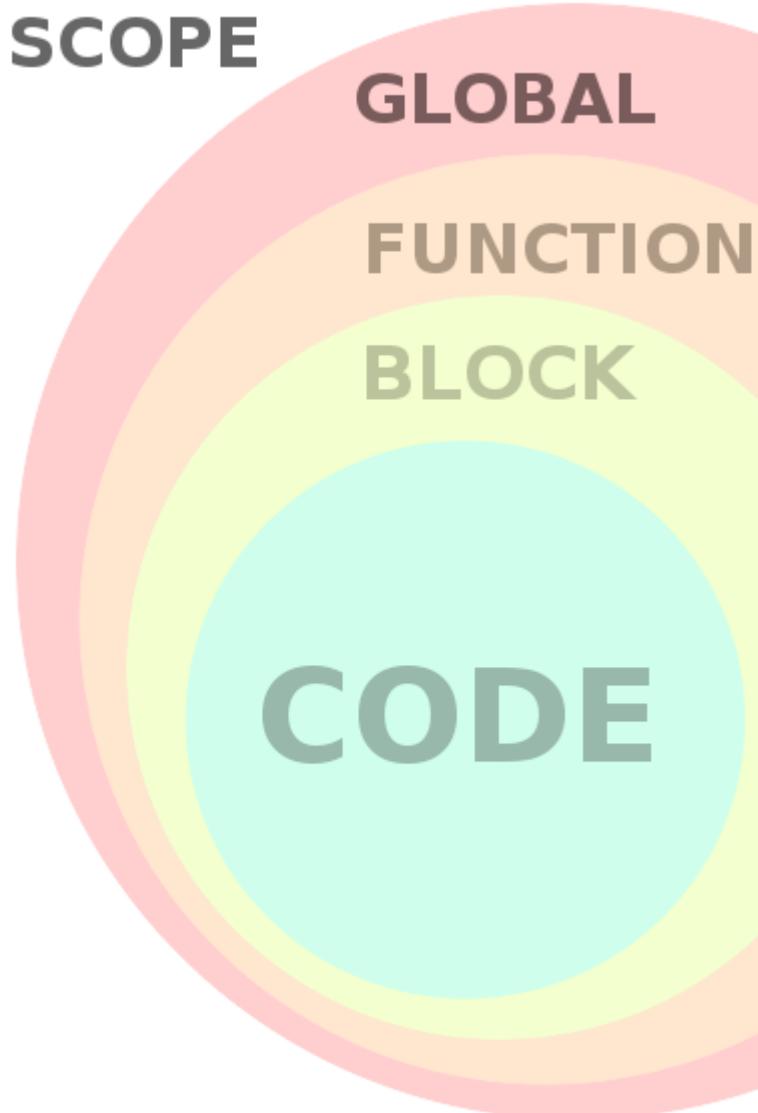
Avec `let` l'affectation se produit après l'initialisation mais avec `const` l'affectation et l'initialisation se produisent ensemble, parce qu'une fois assignée la valeur de const ne peut pas changer,

5. Portée (scope)

La portée (scope) est le contexte d'exécution courant (scope) est le contexte dans lequel les **valeurs** et **expressions** sont "visibles" ou peuvent être référencées.

Si une **variable** ou une **expression** n'est pas dans la portée courante (local scope) , alors son utilisation ne sera pas possible.

Les portées peuvent aussi être empilées hiérarchiquement de manière à ce que les portées enfants puissent accéder aux portées parentes, mais pas l'inverse.

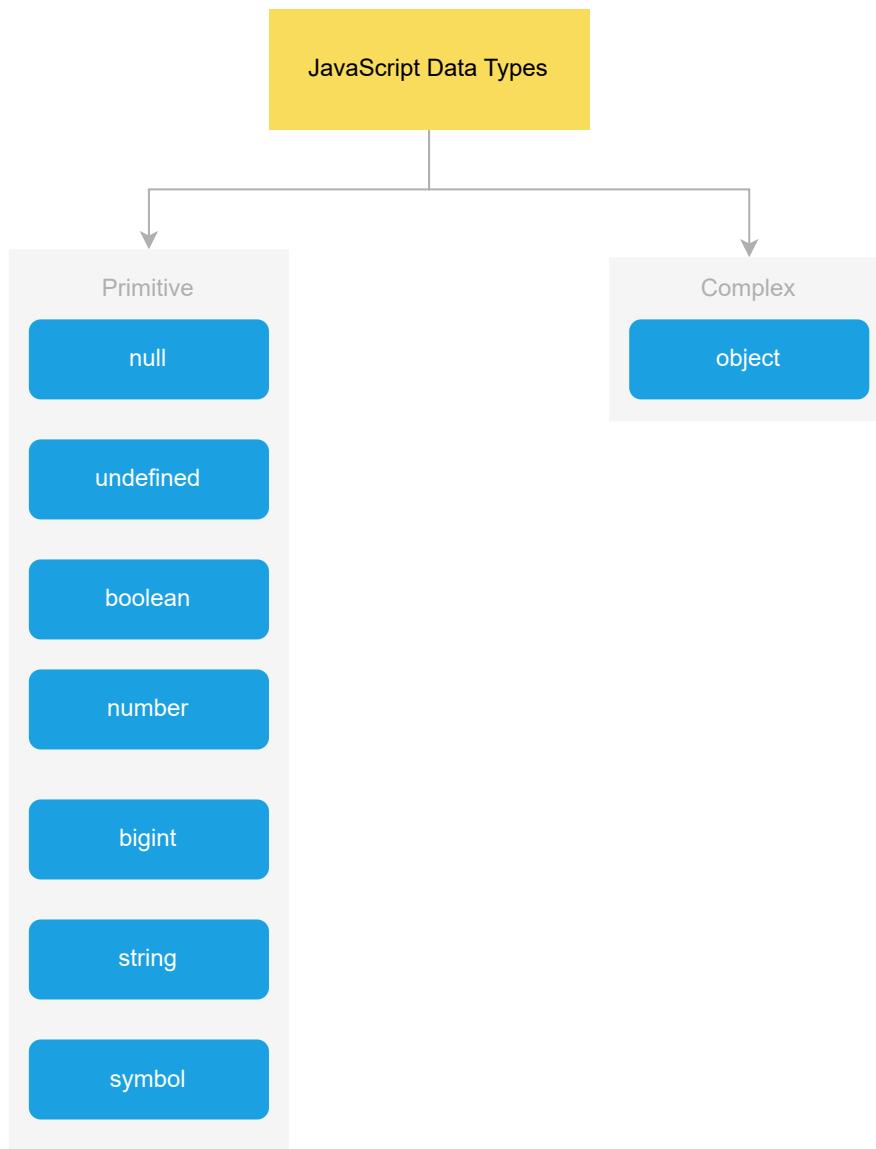


JavaScript dispose des différents types de portée :

- Portée globale : l'étendue par défaut pour tout le code exécuté en mode script.
- Portée de module : L'étendue du code s'exécutant en mode module.
- Portée de fonction : La portée créée avec une fonction.
- Portée de bloc : les variables déclarées avec `let` ou `const` appartiennent à l'étendue créée via une paire d'accolade.

6. Types de données

 `datatype.js`



6.1. Les valeurs primitives

Tous les types, sauf les objets, définissent des valeurs **immuables** (= qu'on ne peut modifier). Les valeurs immuables pour chacun de ces types sont appelées « **valeurs primitives** ».

1. **Number:** représente les nombres comme nous connaissons (entiers et décimaux).
2. **String:** représente une séquence de caractères utilisée pour le texte.
3. **Booléen:** représente **true** ou **false**.
4. **Indéfini:** type de données qui n'a pas encore de valeur.
5. **Null:** type de données qui notifie les points de variable vers aucun. C'est un type de données primitif en JavaScript, même si le résultat de `typeof null` renvoie "`object`". C'est un bug historique dans JavaScript.
6. **NaN :** Représente une valeur "Not-a-Number", souvent résultat d'une opération mathématique invalide.

7. **BigInt:** représente des entiers de grande taille.
8. **Symbol:** Un symbole est une valeur primitive unique et immuable.

6.1.1. Booléens

6.1.1.1. Falsy

- `false`
- `0` (zero)
- `-0` (moins zero)
- `NaN` (`BigInt` zero)
- `''`, `""`, `````` (chaîne vide)
- `null`
- `undefined`
- `NaN`

6.1.1.2. Truthy

- `'0'`
- `'false'`
- `[]` (tableau vide)
- `{}` (objet vide)
- `function(){}()` (fonction vide)

Comparaison avec ==

<code>==</code>	<code>true</code>	<code>false</code>	<code>0</code>	<code>''</code>	<code>null</code>	<code>undefined</code>	<code>NaN</code>	<code>Infinity</code>	<code>[]</code>	<code>{}</code>
<code>true</code>	true	false	false	false	false	false	false	false	false	false
<code>false</code>	false	true	true	true	false	false	false	false	true	false
<code>0</code>	false	true	true	true	false	false	false	false	true	false
<code>''</code>	false	true	true	true	false	false	false	false	true	false
<code>null</code>	false	false	false	false	true	true	false	false	false	false
<code>undefined</code>	false	false	false	false	true	true	false	false	false	false
<code>NaN</code>	false	false	false	false	false	false	false	false	false	false
<code>Infinity</code>	false	false	false	false	false	false	true	false	false	false
<code>[]</code>	false	true	true	true	false	false	false	false	false	false
<code>{}</code>	false	false	false	false	false	false	false	false	false	false

Comparaison avec ===

==	true	false	0	''	null	undefined	NaN	Infinity	[]	{}
true	true	false	false	false	false	false	false	false	false	false
false	false	true	false	false	false	false	false	false	false	false
0	false	false	true	false	false	false	false	false	false	false
''	false	false	false	true	false	false	false	false	false	false
null	false	false	false	false	true	false	false	false	false	false
undefined	false	false	false	false	false	true	false	false	false	false
NaN	false	false	false	false	false	false	false	false	false	false
Infinity	false	false	false	false	false	false	false	true	false	false
[]	false	false	false	false	false	false	false	false	false	false
{}	false	false	false	false	false	false	false	false	false	false

6.2. Les objets

- Comme nous venons de le voir, il existe 8 types de données dans le langage JavaScript.
- Sept d'entre elles sont appelées “primitives”, car leurs valeurs ne contiennent qu'une seule chose (que ce soit une chaîne, un nombre ou autre).
- En revanche, les **objets** sont utilisés pour stocker des **collections de données variées et d'entités plus complexes**. En JavaScript, les objets sont omniprésents dans le langage.

Avec JavaScript, on peut créer un objet à la volée (pas besoin d'avoir recours à des classes ni de créer des instances).

La création d'un objet à la volée est appelée **Object Literal Syntax** ce qui signifie qu'on n'a pas recours à une classe ou à un constructeur pour la création de cet objet.

```
const a = {
    "uid": "abcdef", ————— Property
    "id" : 25
}
      |   |
      Key  Value
```

6.3. Différence entre primitive et objet

 *datatype.js*

6.3.1. Cas des primitives : copie par valeur

- On copie la valeur contenu en mémoire.

```
let a = 10;
let b = a;
b = b + 1;
console.log(a); // 10
console.log(b); // 11
```

**address of a variables created
in memory**

let a = 10 **0x01**
10
 a

let b = a **0x02**
10
 b

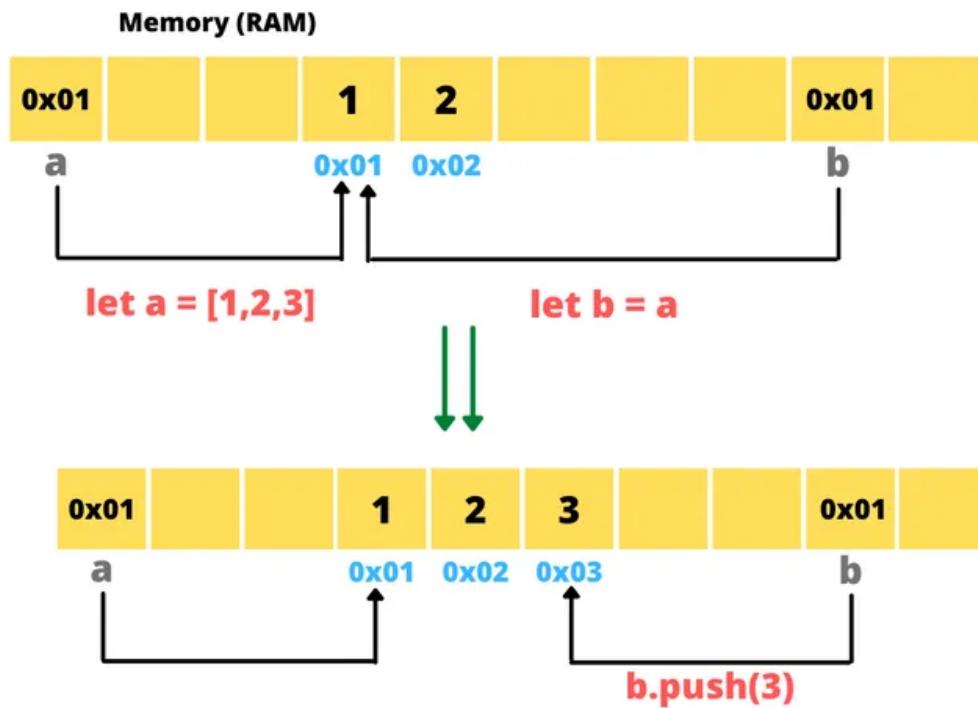
b = b + 1 **0x02**
11
 b

**value inside "a"
remains unchanged
after "b" value changes** **0x01**
10
 a

6.3.2. Cas des objets : copie par référence

- On copie la référence vers l'espace en mémoire.
- Les deux variables pointent alors vers la même valeur.

```
let a = [1,2];
let b = a;
b.push(3);
console.log(a) // [1,2,3]
console.log(b) // [1,2,3]
```



6.3.3. Explication pas à pas

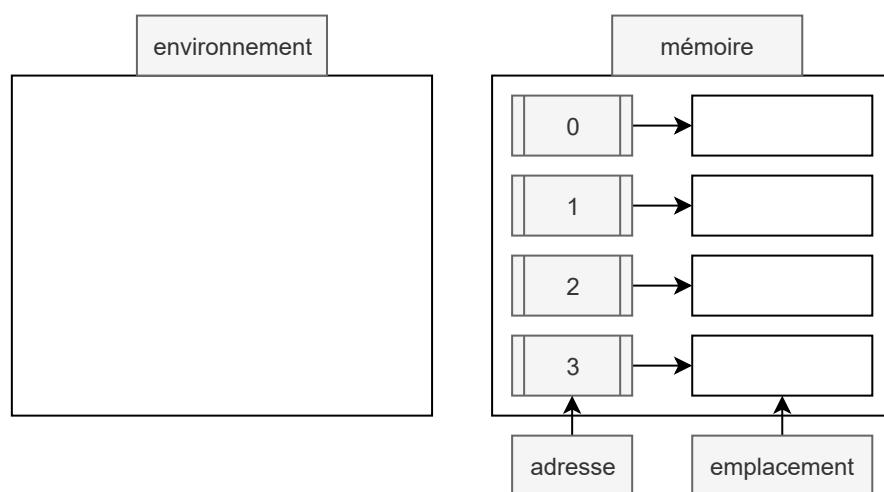
```

let a = 10;
let b = a;
let a = 20;
let c = {}

// créez une variable « b » initialisée avec « a »
var b = a

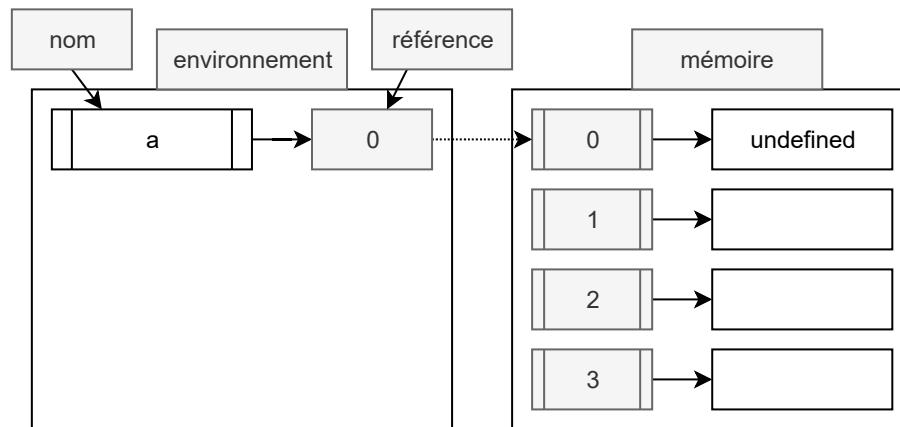
// modifiez l'objet associé à « a »
// on lui ajoute une propriété qqch
// ayant comme valeur 10
a.qqch = 10
  
```

étape 1 : tout est vierge et silencieux



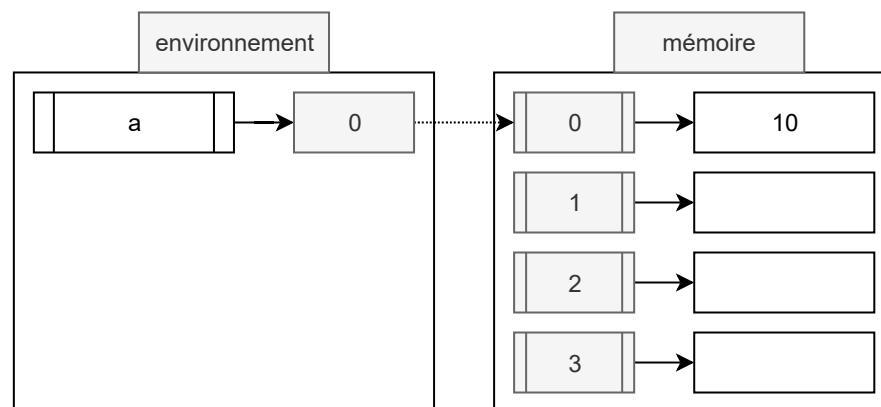
étape 2 : création d'une variable **a** non initialisée dont la valeur est donc **undefined**.

```
let a;
```



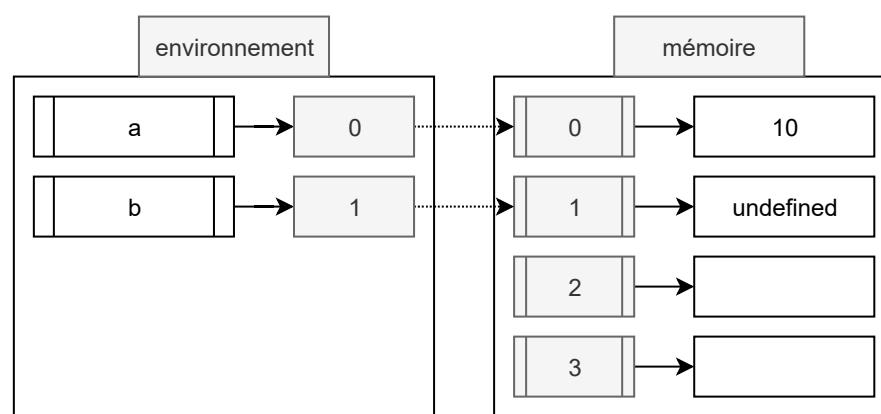
étape 3 : initialisation de **a** avec la valeur 10.

```
a = 10;
```



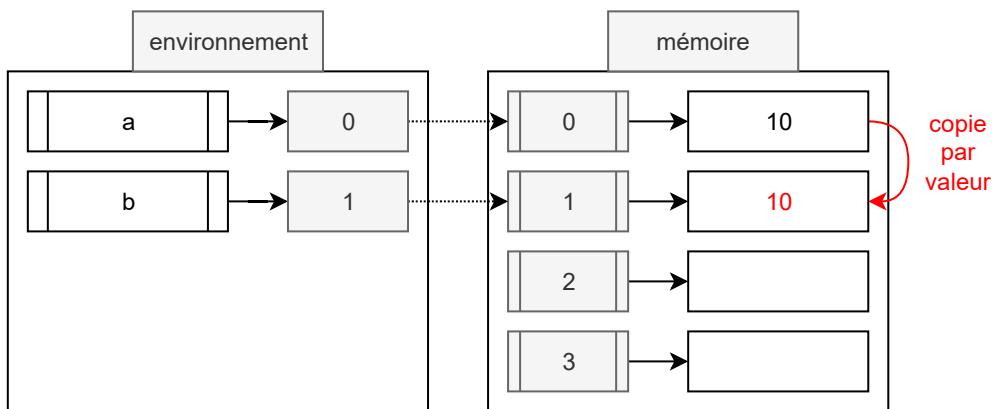
étape 4 : création d'une variable **b** non initialisée dont la valeur est donc **undefined**.

```
let b;
```



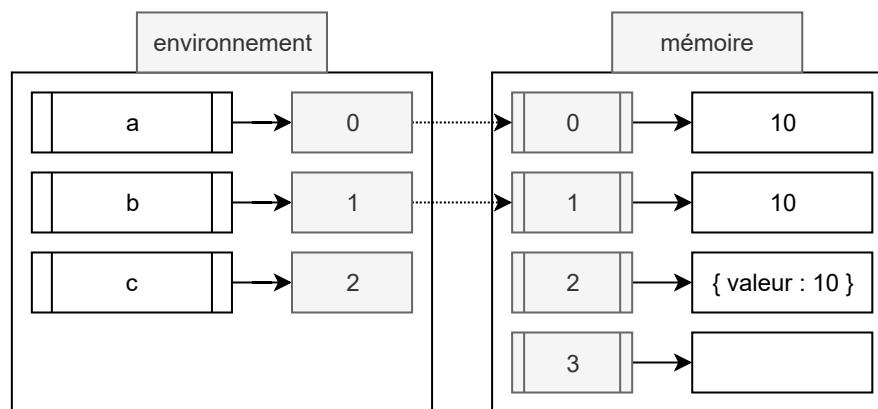
étape 5 : assignation de **b** à la valeur de **a**.

```
let b = a;
```



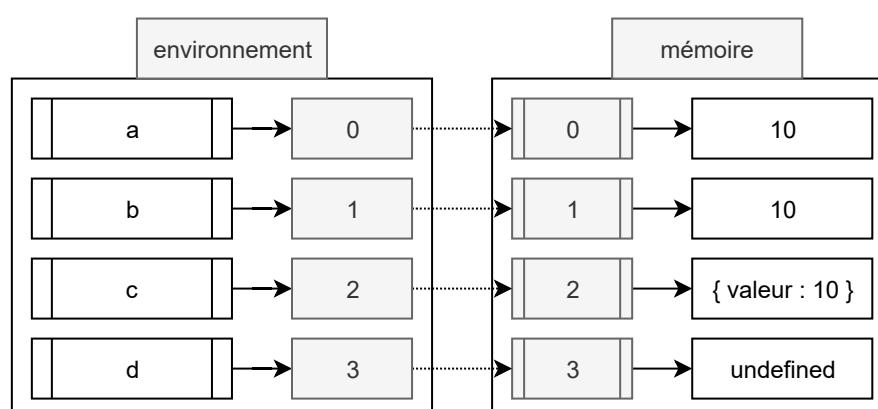
étape 6 : création d'un objet c avec une propriété valeur égale à 10.

```
let c = { valeur : 10 };
```



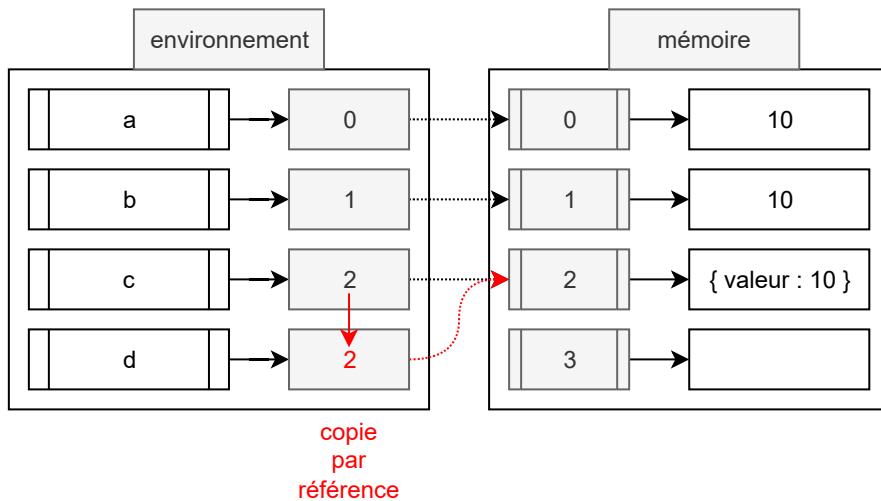
étape 7 : création de la variable d non initialisée et dont la valeur est donc undefined.

```
let d;
```



étape 7 : assignation à d la référence de c.

```
d = c;
```



6.4. Conversion de type et Coercition

conversion_de_type.js

6.5. typeof()

`typeof` est un opérateur qui retourne une chaîne qui indique le type d'une variable (ou d'une expression).

```
typeof 4 // number
typeof 4.5 // number
let name = 'toto'
typeof name // string
typeof 'bonjour' // string
typeof '4' // string
typeof (4 > 5) // boolean
typeof null // object
typeof undefined // undefined
```

6.6. Coercition de type

coercition.js

coercion : Action de contraindre, droit de contraindre quelqu'un à faire quelque chose.

La **coercion de type** fait référence au processus de conversion **automatique** ou **implicite** de valeurs d'un type de données à un autre. Cela inclut la conversion de nombre en string, de string en nombre, de booléen en nombre.

Coercion implicite / explicite :

```
String(123) // explicite
123 + ''    // implicite
```

String :

```
String(123)          // '123'
String(-12.3)        // '-12.3'
String(null)         // 'null'
String(undefined)    // 'undefined'
String(true)          // 'true'
String(false)         // 'false'
```

Boolean :

```
Boolean(2)           // explicite
if (2) { ... }      // implicite car recours à un opérateur logique
!!2                // implicite car recours à un opérateur logique
2 || 'hello'        // implicite car recours à un opérateur logique
```

Valeur numérique :

```
Number('123') // explicite
+'123'        // implicite
123 != '456'  // implicite
4 > '5'       // implicite
5/null        // implicite
true | 0      // implicite
```

Rappel : Lors de l'application de `==` à `null` ou `undefined`, la conversion numérique ne se produit pas. `null` est uniquement égal à null ou `undefined` et n'est égal à rien d'autre.

Finalement :

```
true + false        // 1 car 1 + 0
12 / "6"            // 2
"number" + 15 + 3   // 'number153'
15 + 3 + "number"   // '18number'
[1] > null          // true car '1' > 0
"foo" + + "bar"     // 'fooNaN'
'true' == true      // false
false == 'false'    // false
null == ''          // false
!!"false" == !!"true" // true
['x'] == 'x'        // true
[] + null + 1       // 'null1'
[1,2,3] == [1,2,3]  // false
```

```
{ } + [] + {} + [1]           // '0[object object]1'
! + [] + [] + ! []           // 'truefalse'
new Date(0) - 0              // 0
new Date(0) + 0              // 'Thu Jan 01 1970 02:00:00(EET)0'
```

6.7. A retenir

Catégorie	Type de Données	Exemples/Notes
Primitifs	<code>undefined</code>	Valeur automatique pour les variables non initialisées
	<code>null</code>	Représente une absence délibérée de valeur d'objet
	<code>boolean</code>	<code>true</code> ou <code>false</code>
	<code>number</code>	Inclut <code>Infinity</code> , <code>-Infinity</code> , et <code>Nan</code>
	<code>string</code>	Séquences de caractères
	<code>symbol</code>	Identifiants uniques
	<code>bigint</code>	Pour représenter des entiers très grands
Objets	<code>object</code>	Collections de paires clé/valeur
	<code>array</code>	Listes ordonnées
	<code>function</code>	Blocs de code réutilisables
	<code>Date</code>	Pour manipuler des dates et des heures
	<code>regexp</code>	Expressions régulières
Collections	<code>Map</code>	Paires clé/valeur avec clés de tout type
	<code>Set</code>	Ensemble de valeurs uniques
	<code>WeakMap</code>	Comme <code>Map</code> , mais avec des clés faiblement référencées
	<code>WeakSet</code>	Comme <code>Set</code> , mais avec des membres faiblement référencés
Objets Globaux	<code>Math</code>	Fonctions mathématiques et constantes
	<code>JSON</code>	Parsing et formatage JSON
Types Spéciaux	<code>ArrayBuffer</code>	Pour les données binaires brutes
	<code>TypedArray</code>	Comme <code>Int8Array</code> , <code>Uint32Array</code> , etc.
	<code>Promise</code>	Pour les opérations asynchrones

7. Les objets globaux

7.1. L'objet global String

L'objet `String` gère les chaînes de caractères. Un objet `String` est utilisé afin de représenter et de manipuler une chaîne de caractères.

```
let str = new String("Un objet String");
```

7.2. L'objet global Number

JavaScript n'a qu'un seul type de nombre. `Number` est utilisé pour manipuler les nombres comme des objets. Pour créer un objet `Number`, on utilise le constructeur `Number()`.

```
let a = new Number('123')
```

7.3. L'objet global Math

L'objet `Math` est un objet natif dont les méthodes et propriétés permettent l'utilisation de constantes et fonctions mathématiques. Cet objet n'est pas une fonction. L'objet JavaScript Math vous permet d'effectuer des tâches mathématiques sur des nombres.

Attention : `Math` fonctionne avec le type `Number`. Il ne fonctionne pas avec les grands entiers/[BigInt](#).

```
Math.round(4.6); // 5
```

7.4. L'objet global Date

Les objets JavaScript `Date` représentent un instant donné sur l'axe du temps dans un format indépendant de la plateforme utilisée. Les objets `Date` contiennent un nombre (`Number`) qui représente le nombre de millisecondes écoulées depuis le premier janvier 1970 sur l'échelle UTC.

```
let aujourd'hui = new Date()
```

7.5. L'objet global Array

L'objet global `Array` est utilisé pour créer des tableaux. Les tableaux sont des objets de haut-niveau (en termes de complexité homme-machine) semblables à des listes.

Les tableaux sont des objets semblables à des listes dont le prototype possède des méthodes qui permettent de parcourir et de modifier le tableau.

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

8. Les fonctions (functions)



fonctions.js

Une fonction est une séquence nommée d'`instructions` qui effectue une opération spécifique. Les fonctions sont utilisées pour `décomposer` un programme en `éléments` plus petits et réutilisables.

8.0.1. Déclaration classique

```
function somme(a, b) {
  return a + b;
}
somme(5,10) // 15
```

8.0.2. Fonctions anonymes

Une fonction anonyme est une fonction à laquelle nous avons donné aucun nom.

3 manières d'exécuter une fonction anonyme :

- Enfermer le code de la fonction dans une variable :

```
let show = function () {
  console.log('Bonjour!');
};
show(); // Anonymous function
```

- Auto-invoquer la fonction anonyme (on parle de fonction IFFE = Immediately invoked function execution)

```
(function() {
    console.log('IIFE');
})();
// IIFE
```

- Utiliser un évènement (voir DOM) pour déclencher l'exécution de la fonction :

```
element.addEventListener('click', function(){
    alert('Bonsoir!');
});
```

8.0.3. Expression de fonction

```
const carre = function(nb) { return nb * nb };
const a = carre(4); // 16
```

8.0.4. Fonctions fléchées

Une **expression de fonction fléchée** permet d'avoir une syntaxe plus courte que les expressions de fonctions.

```
// SANS paramètre
const salut = () => 'Bonjour';
salut(); // Bonjour
```

```
// UN SEUL paramètre
const double = a => 2*a;
double(4) // 8
```

```
// DEUX paramètres
const somme = (a, b) => a + b;
somme(5,10) // 15
```

```
// Si la fonction fléchée comporte plusieurs instructions il faut mettre les accolades.
const doubleTriple = (a, b) => {
    console.log(`Le double de ${a} vaut : 2*${a}`);
    console.log(`Le triple de ${b} vaut : 3*${b}`);
}
doubleTriple(10,5)
// Le double de 10 vaut : 20
// Le triple de 5 vaut : 15
```

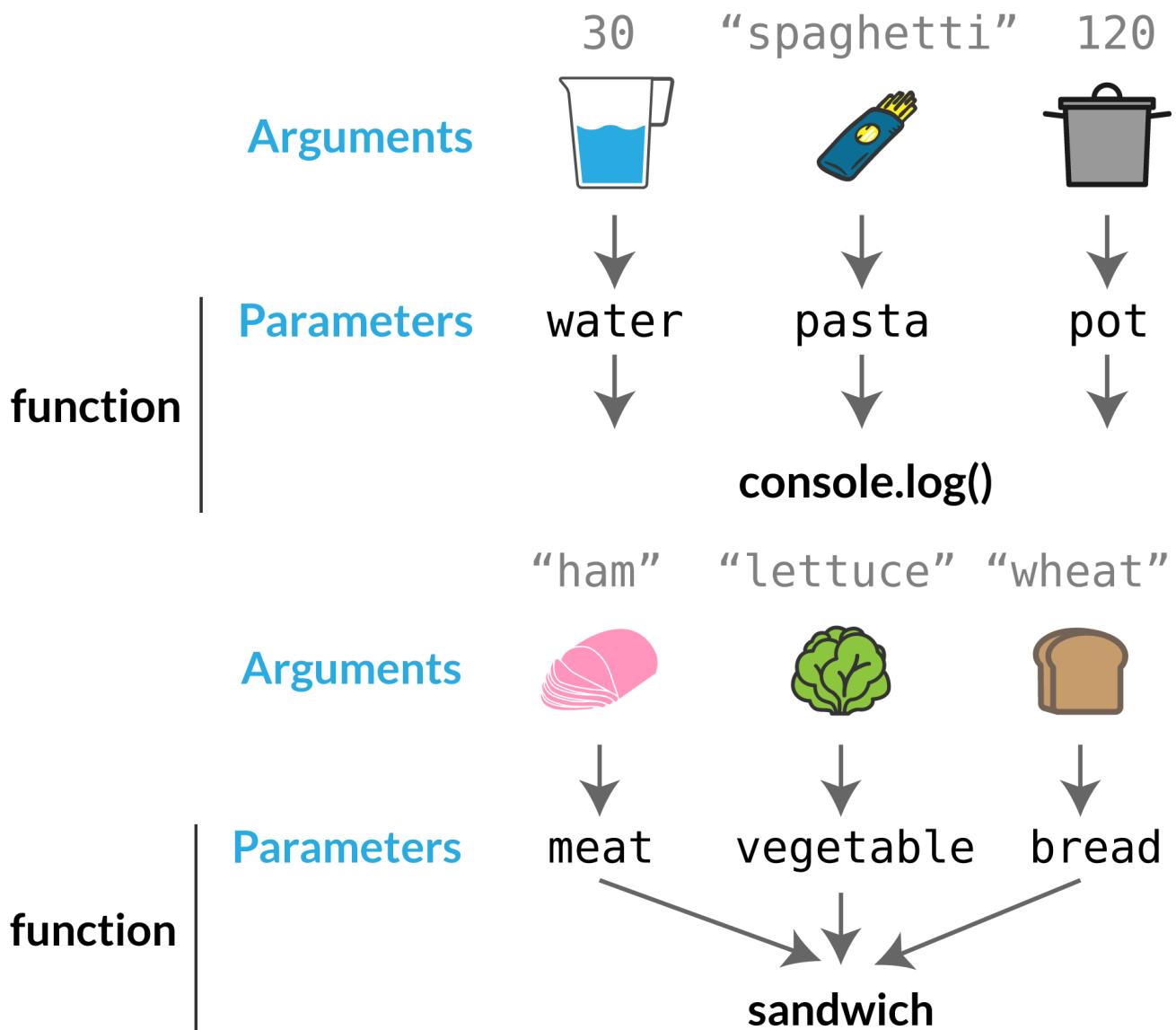
```
function keyword      function name      function parameters  
↓                  ↓                  ↓  
function functionName( parm1, parm2, ... ) {  
    statement1;  
    statement2;  
    statement3; }   function statements  
                    return something; } function return  
                }
```

```
function addition(a,b){  
    return a+b;  
}  
  
addition(5,5);
```

parameter

argument

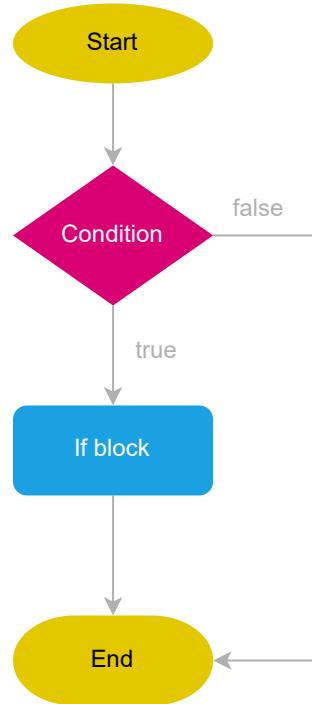
```
function makeSandwich(, , ) {  
    let sandwich =   +  + ;  
    return ;  
}
```



9. Les structures de contrôle

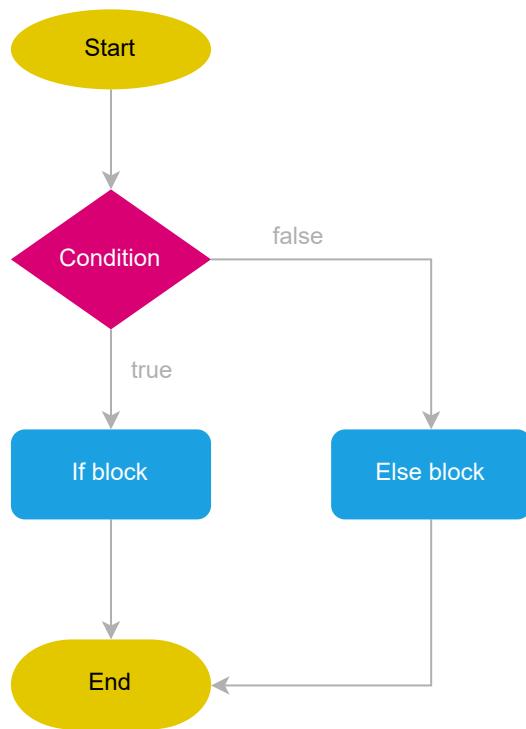
9.1. if

```
if (N > 5) {  
    console.log('La variable N est strictement plus grande que 5');  
}
```

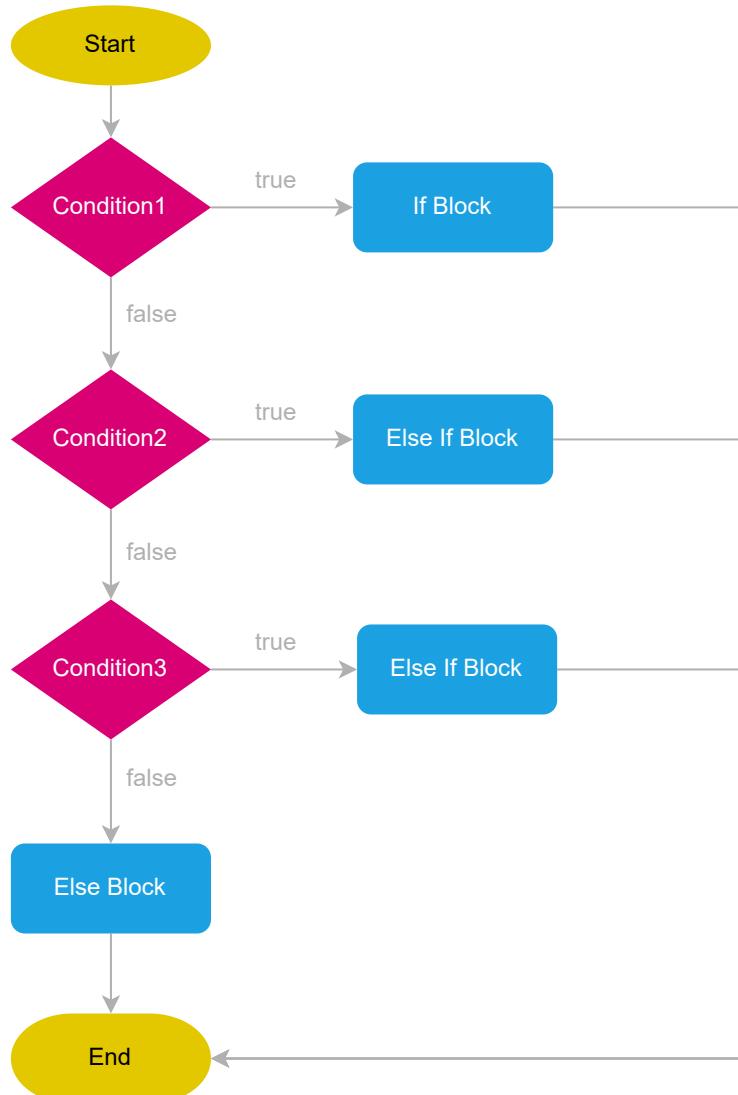


9.2. If ... else

```
if (N > 5) {  
    console.log('La variable N est strictement supérieur à 5');  
} else {  
    console.log('La variable N est strictement inférieur à 5');  
}
```

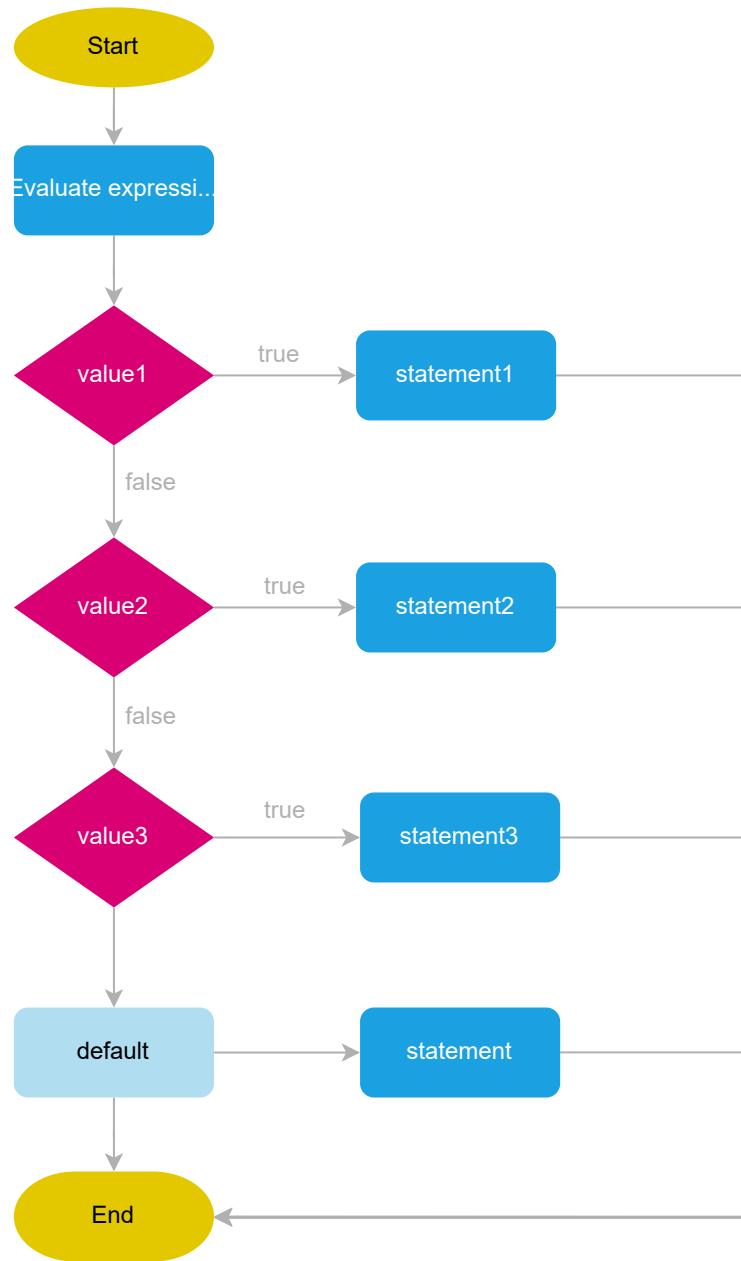


9.3. if ...else ...if



```
if (N > 5) {  
    console.log('La variable N est strictement supérieure à 5');  
} else if (N < 5) {  
    console.log('La variable N est strictement inférieure à 5');  
} else {  
    console.log('La variable N vaut 5');  
}
```

9.4. switch



```

switch(age) {
  case age >= 0 && age <= 3
    texte = "bébé";
    break;
  case age >= 4 && age <= 12:
    texte = "enfant";
    break;
  case age >= 13 && age <= 19:
    texte = "ado";
    break;
  case age >= 20 && age <= 130:
    texte = "adulte";
    break;
  case age < 0 || age > 130 :
    texte = "adulte";
}
  
```

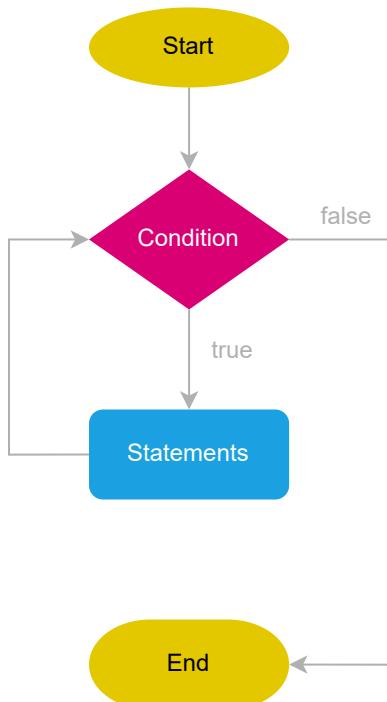
```

break;
default:
  texte = "âge incorrect";
}

```

9.5. while

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



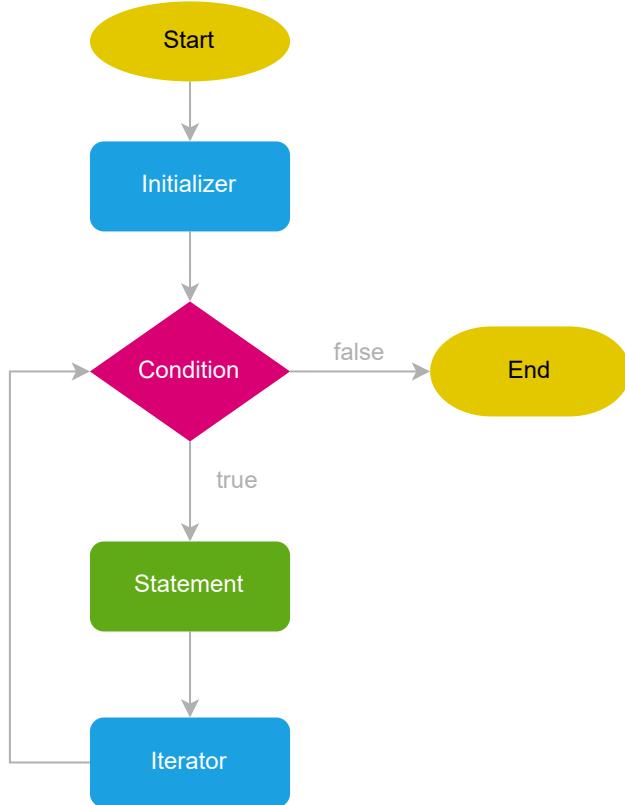
```

let n = 0;
while (n < 3) {
  n++;
}
console.log(n); // 3

```

9.6. do ...while

L'instruction `do...while` crée une boucle qui exécute une instruction jusqu'à ce qu'une condition de test ne soit plus vérifiée. La condition est testée après que l'instruction soit exécutée, le bloc d'instructions défini dans la boucle est donc exécuté au moins une fois.



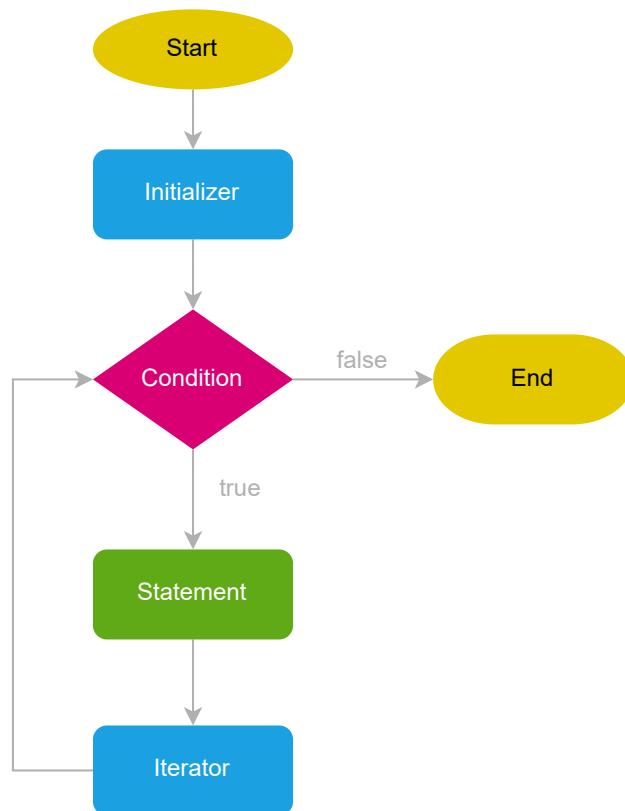
```

let res = '';
let i = 0;

do {
  i = i + 1;    // ou i++
  res = res + i; // ou res++
} while (i < 5);

console.log(res); // res: "12345"
  
```

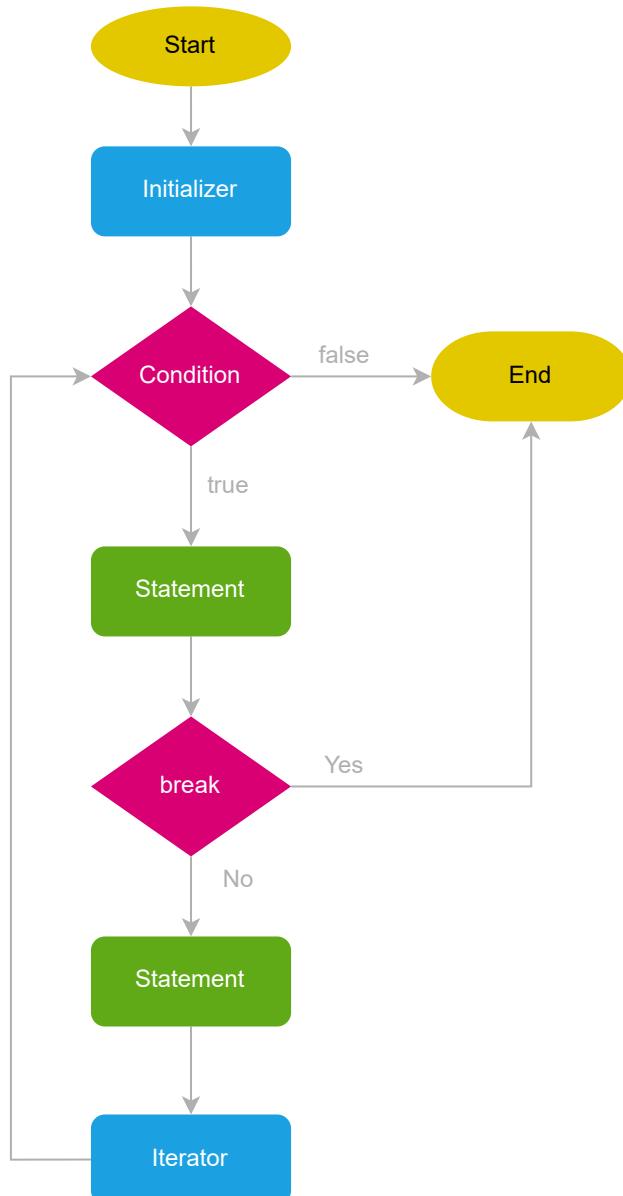
9.7. for



```
txt = '';
for (let i=0; i < 5; i++) {
    txt = txt + i;
}
// 01234
```

9.8. break

L'instruction `break` permet de terminer la boucle en cours et continue d'exécuter le code après la boucle (le cas échéant).

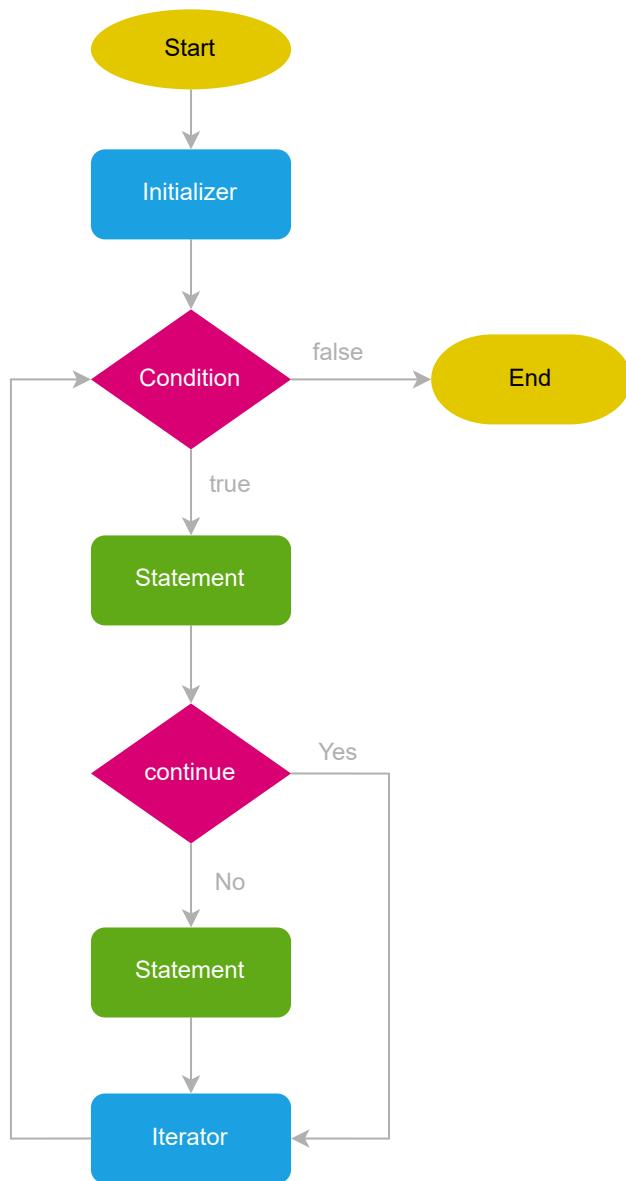


```

let str = "";
for (let i = 0; i < 5; i++) {
  if (i === 3) {
    break;
  }
  str = str + i;
}
// 012
  
```

9.9. continue

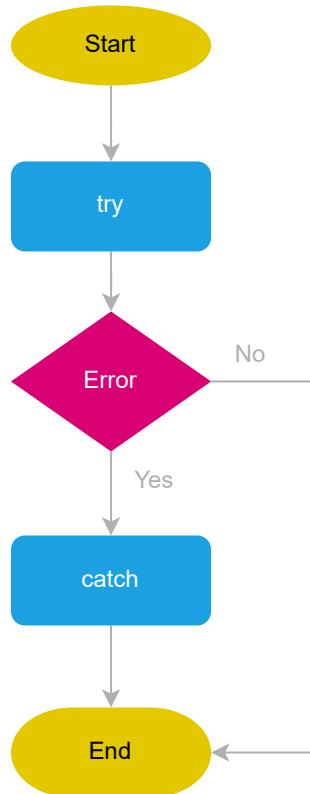
L'instruction `continue` arrête l'exécution des instructions pour l'itération de la boucle courante. L'exécution est reprise à l'itération suivante.



```
let str = "";
for (let i = 0; i < 5; i++) {
  if (i === 3) {
    continue;
  }
  str = str + i;
}
// 01234
```

9.10. try...catch

```
try {
  let result = add(10, 20);
  console.log(result);
} catch (e) {
  console.log({ name: e.name, message: e.message });
}
console.log('Bye');
```



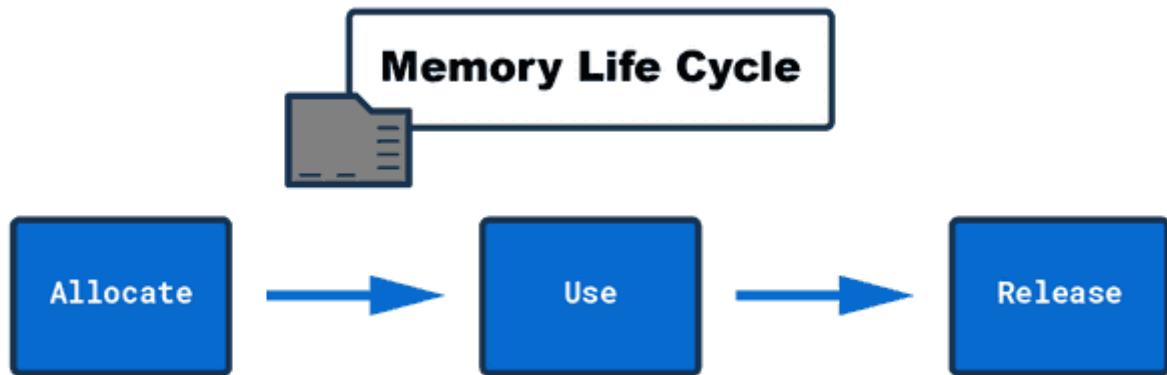
10. Comment JavaScript fonctionne en coulisse ?

10.1. Cycle de vie de la mémoire

En JavaScript, lorsque nous créons des variables, des fonctions ou quoi que ce soit, le moteur JS alloue de la mémoire pour cela et la libère une fois qu'elle n'est plus nécessaire.

- **allocation de mémoire** : processus de réservation d'espace en mémoire.
- **libération de mémoire** libère de l'espace, ainsi prêt à être utilisé à d'autres fins.

Chaque fois que nous attribuons une variable ou créons une fonction, la mémoire pour cela passe toujours par les mêmes étapes suivantes:



- **Allouer de la mémoire** : JavaScript s'en charge pour nous, il alloue la mémoire dont nous aurons besoin pour l'objet que nous avons créé.
- **Utiliser la mémoire** : L'utilisation de la mémoire est quelque chose que nous faisons explicitement dans notre code - lire et écrire en mémoire n'est rien d'autre que lire ou écrire à partir ou vers une variable.
- **Libérer de la mémoire** : Cette étape est également gérée par le moteur JavaScript. Une fois que la mémoire allouée est libérée, elle peut être utilisée à de nouvelles fins.

10.2. La pile (stack) le tas (heap)

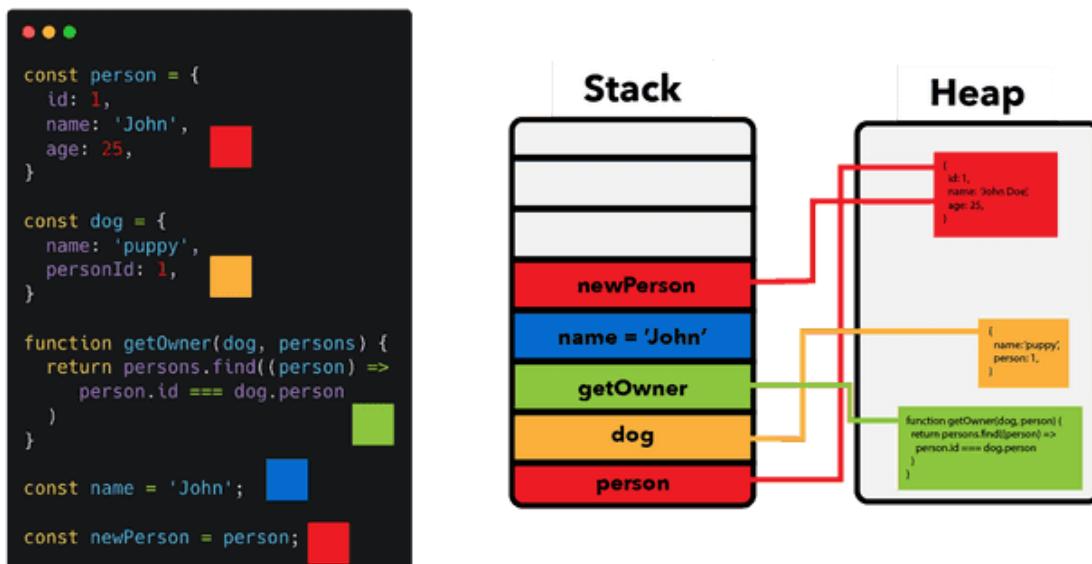
10.2.1. Pile : allocation de la mémoire

Toutes les valeurs sont stockées dans la pile car elles contiennent toutes des valeurs primitives.



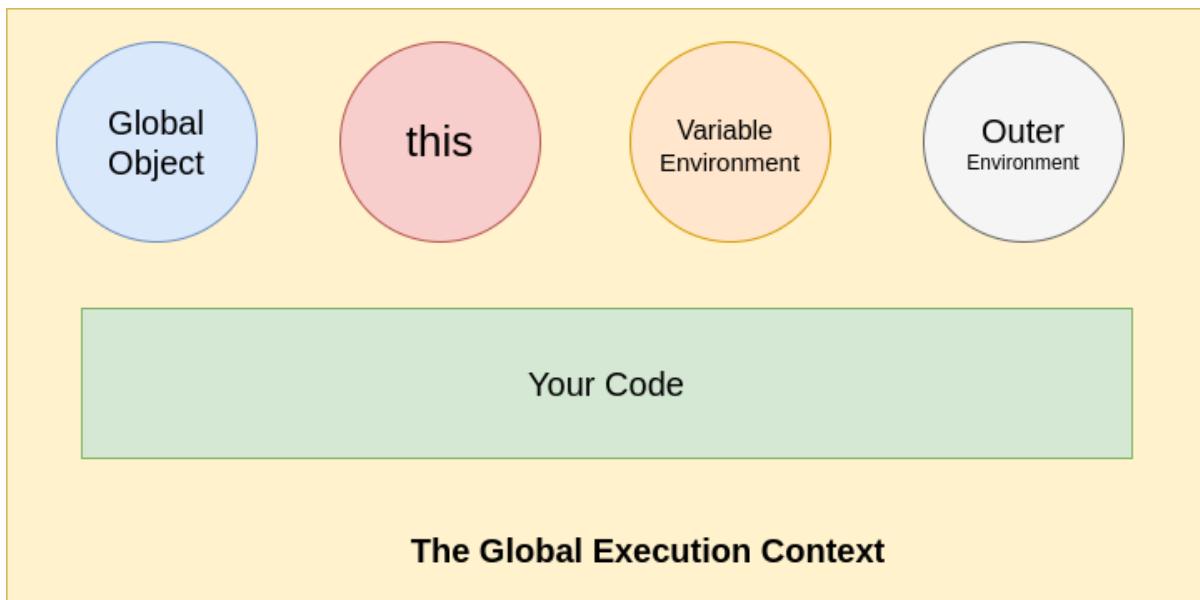
10.2.2. Le tas : allocation de la mémoire dynamique

- Le tas est un espace où JavaScript stocke des **objets** et des fonctions.
- Le moteur **n'alloue pas une quantité fixe de mémoire pour ces objets**.
- Plus d'espace sera alloué au besoin.
- L'allocation de mémoire = **allocation de mémoire dynamique**.



10.3. Contexte d'exécution

En JavaScript, un **contexte d'exécution** est un environnement où le code est évalué et exécuté.



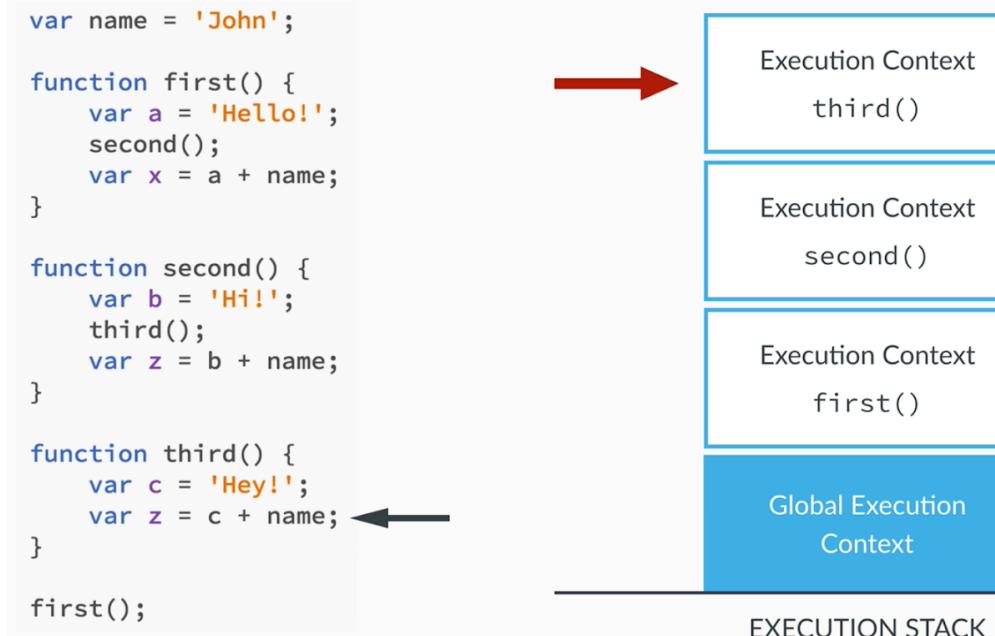
- **Contexte d'Exécution Global :**
 - contexte par défaut où tout le code commence à s'exécuter.
 - Il est créé lorsque le script JavaScript est chargé pour la première fois.
 - Il contient :
 - Les variables globales
 - Les fonctions globales
 - L'objet global (`window` dans un navigateur, `global` dans Node.js)
 - Il n'y a qu'un seul contexte global dans un programme JavaScript.
 - Les variables et les fonctions définies au niveau global sont accessibles partout dans le script.

- **Composants du Contexte d'Exécution Global :**
 - **Global Object (Objet Global) :**
 - En environnement de navigateur, c'est l'objet `window`. En environnement Node.js, c'est l'objet `global`.
 - Contient toutes les variables et fonctions globales.
 - **this :**
 - Dans le contexte global, `this` fait référence à l'objet global (`window` dans les navigateurs).
 - **Variable Environment (Environnement de Variables) :**
 - Contient toutes les variables définies dans le contexte actuel.
 - Dans le contexte global, cela inclut toutes les variables globales.

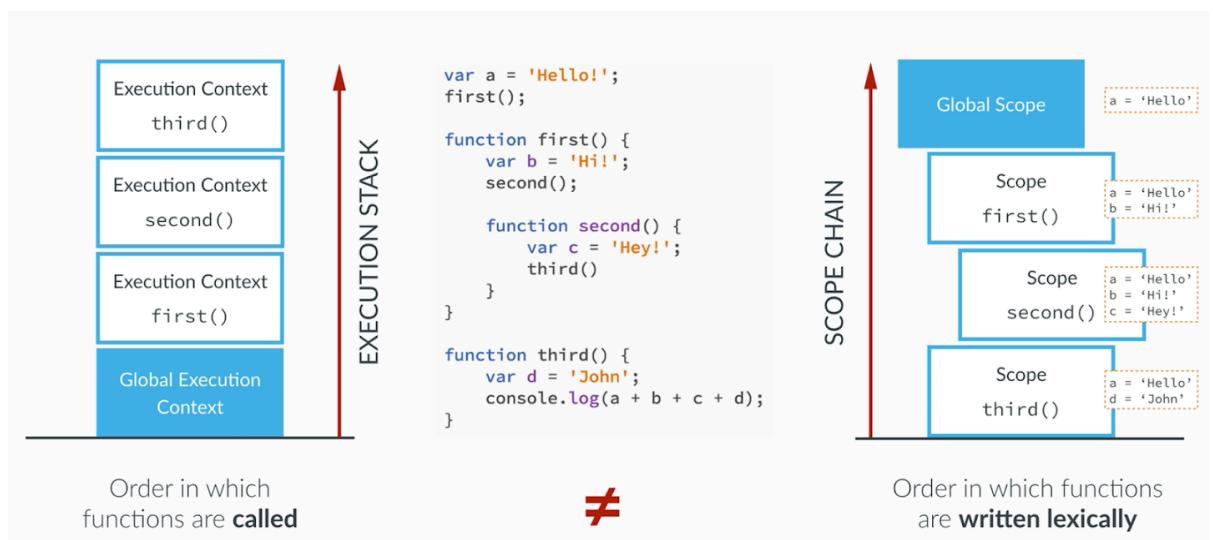
Aspect	Global Execution Context	Global Object
Nature	C'est un environnement d'exécution.	C'est un objet JavaScript.
Contenu	Inclut le Global Object, <code>this</code> , Variable Environment, et Outer Environment.	Contient les variables globales, les fonctions globales et d'autres propriétés globales.
Références	Le premier contexte d'exécution créé lorsqu'un script JavaScript commence à s'exécuter.	Accessible via <code>this</code> dans le contexte global, et directement par des références globales comme <code>window</code> dans un navigateur.

10.4. Contexte d'exécution d'une fonction

Lors de l'exécution du code dans une fonction, un nouveau contexte d'exécution sera créé au-dessus de celui existant.



10.5. Chaîne des portées



10.6. JS RUNTIME

10.6.1. Les différents moteur JS



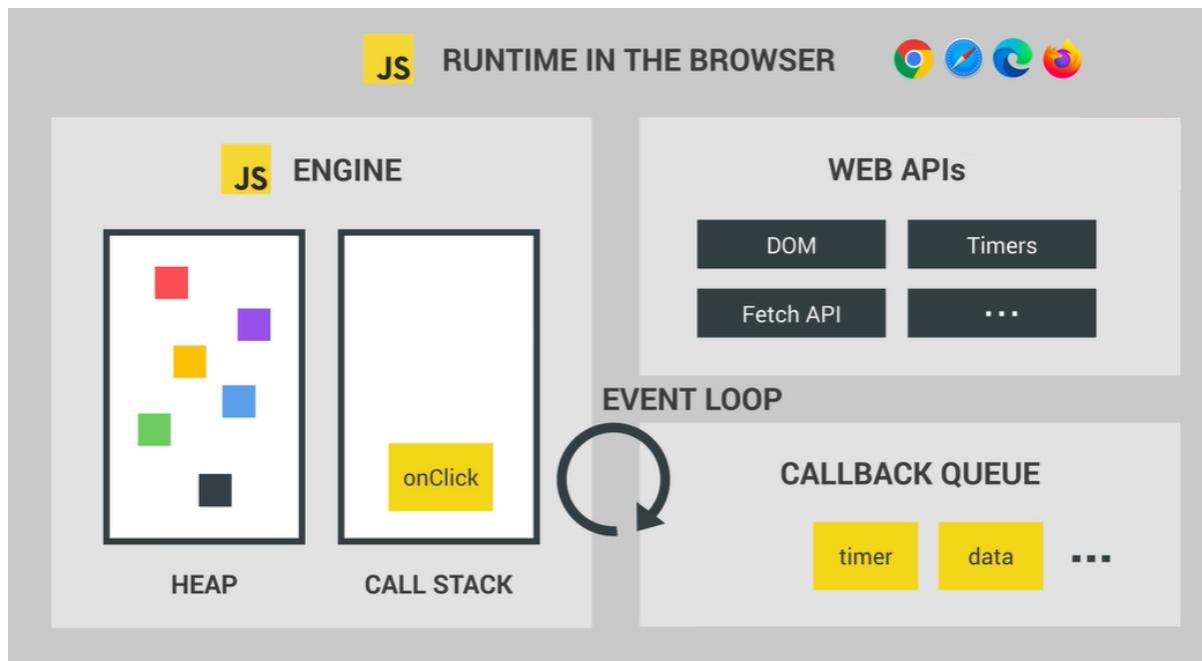
Chakra



SpiderMonkey



v8



10.7. Etapes d'exécution d'un script

