

M2CCI – M2 GEOMAS 2026-2027

cours TW (Technologies du Web)

Introduction au langage JavaScript (2ème partie)

Philippe Genoud

Philippe.Genoud@univ-grenoble-alpes.fr



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Dernière modification : 15/10/2026 16:51

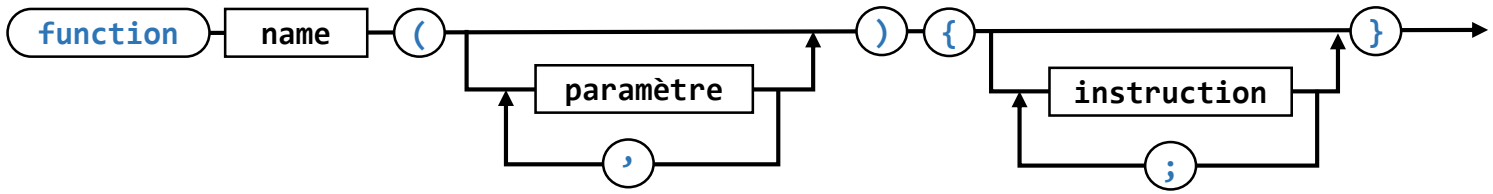
- Fonctions
  - déclaration
  - valeur de retour
  - invocation
  - variables de type fonction (*first class functions*)
  - fonctions d'ordre supérieur (*high order function*)
  - fonctions anonymes
  - fonction fléchées(*arrow functions*)
- Portée (*Scope*) des variables
  - variable locale, variable globale
  - différence entre `let` et `var`
  - Remontée des variables (hoisting)

Fonctions: déclaration

- Déclaration de fonctions similaire à C/C++ sauf :
  - pas de type de retour pour la fonction
  - pas de types pour les paramètres
  - instruction `return` facultative
  - passage des paramètres par valeur uniquement (les paramètres sont une copie des arguments)
- Forme générale

```
function name([parametre1 [, parametre2 [, parametre3 … ]]]) {
        /* code à exécuter */
        …
        [return valeur ;]   // optionnel
}
```

ce qui est entre [et] est facultatif (optionnel)



pour le nommage des fonctions et des paramètres on utilise les même règles que pour les variables.

Exemples de déclarations de fonctions

Fonction sans paramètres

```
function helloWorld() {
    console.log("Hello World");
}
```

Fonction avec un paramètre

```
function hello(nom) {
    console.log("Hello " + nom);
}
```

Fonction avec deux paramètres

```
function helloN(n, nom) {
    let i = 0;
    while (i < n) {
        console.log("Hello " + nom);
        i++;
    }
}
```

Fonction avec deux paramètres et qui renvoie une valeur

```
function nbreOccurrencesLettre(lettre, mot) {
    let nbreOccurences = 0;
    for (let i = 0; i < mot.length; i++) {
        if (mot[i] === lettre) {
            nbreOccurences++;
        }
    }
    return nbreOccurences;
}
```

Fonction : instruction `return`

- instruction **`return`**
  - Permet à la fonction de transmettre une valeur calculée au programme appelant
  - Peut apparaitre zéro, une ou plusieurs fois dans le corps de la fonction
  - La valeur retournée peut être la valeur d'une expression de n'importe quel type
    - `boolean`, `number`, `string`, `null`, `undefined`, `object`

Exemple : écrire une fonction qui cherche si un nombre n est premier

Un nombre n est premier si il n'a pas d'autre diviseur que 1 et lui même

boucle recherche un nombre d_1 qui est le plus petit diviseur de n autre que n, c'est-à-dire un nombre $d_1$pour lequel il existe un nombre d_2 ($d_1 < d_2 < n$) tel que $d_1 \times d_2 = n$. Si aucun nombre $\leq \sqrt{n}$ n'est diviseur de n, inutile d'aller plus loin en effet $d_2$étant plus grand que $d_1 = \sqrt{n}$ le produit $d_1 \times d_2$ sera nécessairement $> n$

```
/**
 * Teste si un nombre est premier ou non
 * @param {number} n le nombre à  tester (on suppose que n > 0)
 *
 * @returns {boolean} true si n est premier, false sinon
 */
function estPremier(n) {
  let resultat;
  if (n < 2) {
    resultat = false;
  } if (n == 2) {
    resultat = true;
  } else {
    let d1 = 2;
    while (d1 <= Math.sqrt(n) && n % d1 !== 0) {
        // n n'est pas divisible par d1
        d1++
    }
    resultat = n % d1 !== 0;
  }
  return resultat;
}
```

`Math.sqrt(x)` (*square root*) calcule la racine carrée d'un nombre.

Équivalent à

```
if (n % d1 !== 0) {
     resultat = true;
}  else {
      resultat = false;
}
```

⇔

La même fonction avec un **`return`** pour chacun des cas

```
/**
 * Teste si un nombre est premier ou non
 * @param {number} n le nombre à  tester (on suppose que n > 0)
 *
 * @returns {boolean} true si n est premier, false sinon
 */
function estPremier(n) {
  if (n < 2) {
    return false;
  } if (n == 2) {
    return true;
  } else {
    for (let d1 = 2; d1 <= Math.sqrt(n); d1++) {
      if (n % d1 === 0) {  // n est divisible par d1
        return false; // il n'est pas premier
      }
    }
    // on a atteint la fin de boucle sans trouver de diviseur
    // n est premier
    return true;
  }
}
```

Fonctions : valeur de retour

- Pas obligatoire de spécifier une valeur de retour une fonction
  - cependant toutes les fonctions retournent une valeur

  →**`undefined`** si pas d'instruction **`return`** dans la fonction ou si **`return`** seul sans valeur de retour;
  - attention aux erreurs si votre code n'est pas capable de gérer ce type de valeur
  - c'est une bonne pratique de spécifier une valeur de retour (par ex. **`false`** ou **`this`** *)

* on reparlera de **`this`** lorsque les objets seront abordés plus en détail



- Attention JavaScript à un mécanisme auto-correcteur* qui peut être un piège !

```
function hello(message) {
     return
          "Hello " + message;
}

console.log(hello("World"));  -------->   undefined
```

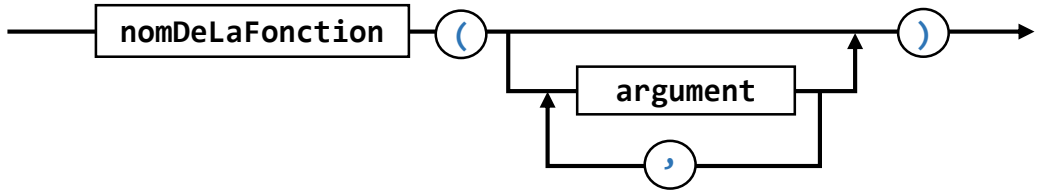
un `;` est ajouté ici !



*Automatic Semicolon Insertion (ASI) : pour en savoir plus https://eslint.org/docs/rules/semi

Fonctions: invocation

- similaire à C/C++
- forme générale




```
nomDeLaFonction([argument1 [, argument2 [, argument3 … ]]])
```

- la liste des arguments est mise en correspondance avec la liste des paramètres
  - argument1 → paramètre1
  - argument2 → paramètre2
  - …
- les arguments sont des **expressions qui sont évaluées**, leur valeur est affectée au paramètre qui correspond à une variable locale à la fonction (**passage par valeur**).

exemples

```
function helloWorld() {
    console.log("Hello World");
}
```

```
function helloN(n, nom) {
    let i = 0;
    nom = nom.toUpperCase();
    while (i < n) {
        console.log("Hello " + nom);
        i++;
    }
}
```

```
function nbreOccurrencesLettre(lettre, mot) {
    let nbreOccurences = 0;
    for (let i = 0; i < mot.length; i++) {
        if (mot[i] === lettre) {
            nbreOccurences++;
        }
    }
    return nbreOccurences;
}
```

```
helloWorld();
```

```
Hello World
```

```
helloN(3, "Winter");
```

```
Hello WINTER
Hello WINTER
Hello WINTER
```

```
let prenom = "Jean";
let nom = "Dupont";
let nb = 1;
helloN(2 * nb + 1, `${prenom} ${nom}`);
```

```
Hello JEAN DUPONT
Hello JEAN DUPONT
Hello JEAN DUPONT
```

```
let s = "abracadabra";
console.log(`la lettre a est présente\
 ${nbreOccurrencesLettre("a", s)}\
 fois dans ${s}`);
```

```
la lettre a est présente 5 fois dans abracadabra
```

Fonctions : invocation

- Passage des paramètres par valeur

```
function helloN(n, nom) {
    let i = 0;
    nom = nom.toUpperCase();
    while (i < n) {
        console.log("Hello " + nom);
        i++;
    }
}
```

```
let nom = "Elodie";
helloN(2, nom);
console.log(nom);
```

```
Hello ELODIE
Hello ELODIE
Elodie
```

1

```
1  function helloN(n, nom) {
2    let i = 0;
3    nom = nom.toUpperCase();
4    while (i < n) {
5      console.log("Hello " + nom);
6      i++;
7    }
8  }
9
10 let nom = "Elodie";
11 helloN(2, nom);
12 console.log(nom);
```

Contexte d'exécution : ensemble des variables accessibles à l'instruction en cours d'exécution

pgm principal

nom	"Elodie"

2

Quand une fonction est appelée les variables définissant les paramètres de la fonction plus les variables déclarées dans la fonction (variables locales) sont ajoutées au contexte d'exécution

Les variables correspondant aux paramètres sont initialisées avec les valeurs des arguments

2

```
1  function helloN(n, nom) {
2    let i = 0;
3    nom = nom.toUpperCase();
4    while (i < n) {
5      console.log("Hello " + nom);
6      i++;
7    }
8  }
9
10 let nom = "Elodie";
11 helloN(2, nom);
12 console.log(nom);
```

helloN

n	2
nom	"Elodie"
i	undefined

pgm principal

nom	"Elodie"

Quand une variable est accédée elle est recherchée dans le contexte d'exécution en partant du haut vers le bas (pile des appels (*call stack*))

(3) `nom = nom.toUpperCase();`

3

```
1  function helloN(n, nom) {
2    let i = 0;
3    nom = nom.toUpperCase();
4    while (i < n) {
5      console.log("Hello " + nom);
6      i++;
7    }
8  }
9
10 let nom = "Elodie";
11 helloN(2, nom);
12 console.log(nom);
```

Ici **nom** est la variable locale au contexte de **helloN**

nom

helloN

n	2
nom	"ELODIE"
i	0

pgm principal

nom	"Elodie"

Quand l'exécution d'une fonction est terminée son contexte est effacé (dépilé de la pile des appels)

(12) `console.log(nom);`

4

```
1  function helloN(n, nom) {
2    let i = 0;
3    nom = nom.toUpperCase();
4    while (i < n) {
5      console.log("Hello " + nom);
6      i++;
7    }
8  }
9
10 let nom = "Elodie";
11 helloN(2, nom);
12 console.log(nom);
```

Ici **nom** est la variable associée au contexte du programme appelant dont la valeur n'a pas été modifiée par la fonction

nom

pgm principal

nom	"Elodie"

© UGA-2026-2027 Philippe GENOUD - M2 CCI - M2 GEOMAS - Cours Technos Web

Fonctions: invocation

- pas de restrictions sur les arguments que l'on peut passer à une fonction
  - on peut passer n'importe quel objet, variable ou valeur à une fonction



```
1   function helloN(n, nom) {
2     let i = 0;
3     nom = nom.toUpperCase();
4     while (i < n) {
5       console.log("Hello " + nom);
6       i++;
7     }
8   }
9
10  let nom = "Elodie";
11  helloN(2, nom);
12  console.log("---");
13  helloN("2", nom);
14  console.log("---");
15  helloN("5z", nom);
16  console.log("---");
17  let elo = {
18    nom: "DUPONT",
19    prenom: "Elodie",
20  };
21  helloN(2, elo);
22  console.log("---");
23  helloN(2, elo.nom);
```

```
PS P:\ExempleCoursJS> node .\parametresQuelconques.js
Hello ELODIE
Hello ELODIE
---
Hello ELODIE
Hello ELODIE
---
---
P:\ExempleCoursJS\parametresQuelconques.js:3
  nom = nom.toUpperCase();

TypeError: nom.toUpperCase is not a function
    at helloN (P:\ExempleCoursJS\parametresQuelconques.js:3:13)
    at Object.<anonymous> (P:\ExempleCoursJS\parametresQuelconques.js:21:1)
    at Module._compile (node:internal/modules/cjs/loader:1099:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1153:10)
    at Module.load (node:internal/modules/cjs/loader:975:32)
    at Function.Module._load (node:internal/modules/cjs/loader:822:12)
    at Function.executeUserEntryPoint [as runMain]
(node:internal/modules/run_main:77:12)
    at node:internal/main/run_main_module:17:47

Node.js v17.7.2
PS P:\ExempleCoursJS>
```

La variable **n** est de type **number**. Dans le test ligne 4 → 2 itérations.

La variable **n** est de type **string**. Dans le test ligne 4 elle est convertie **number** → 2 itérations.

La variable **n** est de type **string**. Dans le test ligne 4 elle est convertie **number** → **NaN** car sa valeur **"5z"** ne correspond pas à un nombre. Le test ligne 4 (**i <n**) est faux . On ne rentre pas dans la boucle

Le paramètre **nom** est un objet, il n'a pas de méthode **toUpperCase** qui est une méthode du type **string** d'où l'erreur d'exécution

l'erreur précédente a interrompu l'exécution du programme. Les instructions ligne 22 et 23 ne sont pas exécutées

Fonctions: invocation

- pas de controles sur les arguments que l'on peut passer à une fonction
  - on peut passer n'importe quel objet, variable ou valeur à une fonction
  - on peut passer plus d'arguments que ce que la fonction attend: ils sont ignorés
  - on peut passer moins d'arguments que ce que la fonction attend: les paramètres manquant seront automatiquement initialisés à **undefined**



parametresExtrasOuManquant.js > ...

```
1   function helloN(n, nom) {
2     let i = 0;
3     while (i < n) {
4       console.log("Hello " + nom);
5       i++;
6     }
7   }
8   helloN(3, "Elodie", "Audrey", "Mathieu");
9   console.log("---");
10  helloN(2);
```

```
PS P:\ExempleCoursJS> node .\parametresExtrasOuManquant.js
Hello Elodie
Hello Elodie
Hello Elodie
---
Hello undefined
Hello undefined
PS P:\ExempleCoursJS>
```

Les arguments supplémentaires **"Audrey"** et **"Mathieu"** sont ignorés.

Il manque l'arguments correspondant au paramètre **nom** sa valeur est initialisée avec **undefined**

parametresDefault1.js > ...

```
1   function helloN(n, nom) {
2     let i = 0;
3     if (! nom) {
4       nom = "World";
5     }
6     while (i < n) {
7       console.log("Hello " + nom);
8       i++;
9     }
10  }
11
12  helloN(2);
13  console.log("---");
14  helloN(2, "");
```

```
> node .\parametresDefault1.js
Hello World
Hello World
---
Hello World
Hello World
PS P:\ExempleCoursJS>
```

1er appel : nom prend la valeur **undefined** qui dans un contexte booléen est évaluée à **false**, le test ligne 3 est donc **true** on utilise la valeur "**World**".

2ème appel :c'est pareil, dans un contexte booléen la chaîne vide est évaluée à **false**.

- Pour les arguments manquant possibilité de définir une valeur par défaut si un paramètre a la valeur **undefined**

parametresDefault2.js > ...

```
1   function helloN(n = 1, nom = "World") {
2     let i = 0;
3     while (i < n) {
4       console.log("Hello " + nom);
5       i++;
6     }
7   }
8
9   helloN(2);
10  console.log("---");
11  helloN(2, "");
12  console.log("---");
13  helloN();
```

```
> node .\parametresDefault2.js
Hello World
Hello World
---
Hello
Hello
---
Hello World
PS P:\ExempleCoursJS>
```

Utilisation de la valeur par défaut pour **nom**

nom a pour valeur **""**

Utilisation des valeurs par défaut pour **n** et **nom**

Fonctions: invocation

```
1  <!DOCTYPE html>
2  <html lang="en">
3
4  <head>
5      <meta charset="UTF-8">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <meta http-equiv="X-UA-Compatible" content="ie=edge">
8      <title>Document</title>
9  </head>
10
11 <body>
12     <h1>Exemple de fonction</h1>
13     <script>
14         let nb = prompt("Entrez un nombre : ", "");
15         nb = parseInt(nb);
16         document.write("le nombre " + nb +
17             " est premier : " + estPremier(nb));
18     </script>
19     <script src="nbresPremiers.js"></script>
20 </body>
21 </html>
```

Dans une page web, l'ordre dans lesquels les scripts apparaissent a de l'importance

Erreur car la fonction est exécutée avant que le moteur JavaScript ait rencontré sa déclaration. Il ne la connait donc pas.

message d'erreur dans la console JavaScript du navigateur

```
▸ Uncaught ReferenceError: estPremier is not defined
    at ex1fonctions.1.html:16
```

Ce script contient la déclaration de la fonction `estPremier()`

ex1fonctions.1.html

Fonctions: invocation

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <meta http-equiv="X-UA-Compatible" content="ie=edge">
7      <title>Document</title>
8      <script src="nbresPremiers.js"></script>
9  </head>
10 <body>
11     <h1>Exemple de fonction</h1>
12     <script>
13       let nb = prompt("Entrez un nombre : ", "");
14       nb = parseInt(nb);
15       document.write("le nombre " + nb +
16                      " est premier : " + estPremier(nb));
17     </script>
18 </body>
19 </html>
```

Dans une page web, l'ordre dans lesquels les scripts apparaissent a de l'importance

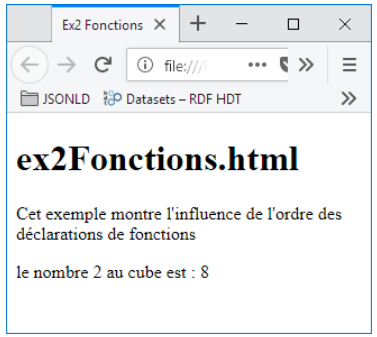
Pour pouvoir être invoquée la déclaration de fonction doit au préalable avoir été chargée. D'où par exemple ici son insertion dans l'élément <head>

ex1fonctions.html

Fonctions: invocation et ordre des déclarations

- dans un script l'ordre de déclaration des fonctions n'a pas d'importance

ex2fonctions.html

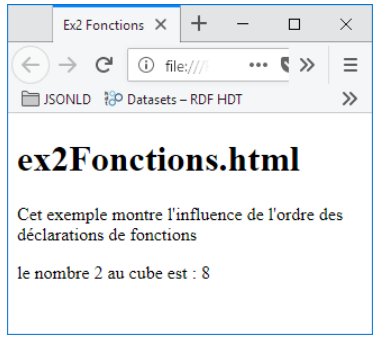



```
function square(x) {
    return x * x;
}

function cube(x) {
    return x * square(x);
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
       content="width=device-width, initial-scale=1.0">
    <title>Ex2 Fonctions</title>
    <script src="ex2Fonctions.js"></script>
</head>
<body>
    <h1>ex2Fonctions.html</h1>
    <p>Cet exemple montre l'influence de l'ordre
        des déclarations de fonctions</p>
    <p>
        <script>
            let nb = 2;
            document.write("le nombre " + nb +
                " au cube est : " + cube(nb));
        </script>
    </p>
</body>
</html>
```

ex2fonctions.1.html




```
function cube(x) {
    return x * square(x);
}

function square(x) {
    return x * x;
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
       content="width=device-width, initial-scale=1.0">
    <title>Ex2 Fonctions</title>
    <script src="ex2Fonctions.1.js"></script>
</head>
<body>
    <h1>ex2Fonctions.html</h1>
    <p>Cet exemple montre l'influence de l'ordre
        des déclarations de fonctions</p>
    <p>
        <script>
            let nb = 2;
            document.write("le nombre " + nb +
                " au cube est : " + cube(nb));
        </script>
    </p>
</body>
</html>
```

Fonctions: function expressions

- En JavaScript les fonctions sont des objets* → on peut affecter une fonction à une ou plusieurs variables

```
function helloWorld(nb) {
   for (let i=0; i < nb; i++) {
       console.log("Hello World !");
   }
} 
```

Pas de point virgule, c'est une déclaration

affectation à partir d'une fonction existante

```
let f1 = helloWorld;
let f2 = f1;

                  // équivalent à helloWorld(3)
f1(3);
                  // équivalent à helloWorld(1)
f2(1);
```

```
Hello World !
Hello World !
Hello World !
Hello World !
```

expression *function* : affectation à partir d'une fonction anonyme

```
let f3 = function(message,nb) {
    for (let i=0; i < nb; i++) {
       console.log(message + " World !");
    }
} ;
```

Point virgule pour marquer la fin de l'instruction d'affectation

```
f3("Bye Bye", 2);
```

```
Bye Bye World !
Bye Bye World !
```

** on reviendra plus en detail sur ce point plus tard.*

© UGA-2026-2027 Philippe GENOUD - M2 CCI - M2 GEOMAS - Cours Technos Web

Fonctions paramètres d'une fonction – *callback*

- une fonction peut être passée en paramètre d'une autre fonction

```
function traiterTableau(tab, operation) {
    for (let i = 0; i < tab.length; i++) {
      tab[i] = operation(tab[i]);
    }
}

function ajoute10(x) {
   return x + 10;
}

let tab1 = [1,2,3,4,5];
traiterTableau(tab1, ajoute10);
console.log(tab1); ----------> [11, 12, 13, 14, 15]

traiterTableau(tab1, function(x) {
      return x * 10;
   }
);
console.log(tab1); ----------> [110, 120, 130, 140, 150]
```

Le paramètre **operation** est une fonction

Cette fonction sera appelée (exécutée) plus tard lorsque la fonction **traiterTableau** sera exécutée : d'où le nom de fonction **callback**

L'argument pour le paramètre **operation** est la fonction **ajoute10**

L'argument pour le paramètre **operation** est une **expression fonction** (**ici** fonction anonyme qui ne sera pas accessible en dehors de cet appel à **traiterTableau**)

Fonction fléchées (Arrow functions)

- Introduites avec ES6 (EcmaScript 2015)
  - Permettent un écriture plus directe de fonctions simples

```
function traiterTableau(tab, operation) {
    for (let i = 0; i < tab.length; i++) {
      tab[i] = operation(tab[i]);
    }
}

function ajoute10(x) {
   return x + 10;
}

let tab1 = [1,2,3,4,5];
traiterTableau(tab1, ajoute10);
console.log(tab1);

traiterTableau(tab1, function(x) {
      return x * 10;
   }
);
console.log(tab1);
```

```
function traiterTableau(tab, operation) {
    for (let i = 0; i < tab.length; i++) {
      tab[i] = operation(tab[i]);
    }
}


let tab1 = [1,2,3,4,5];
traiterTableau(tab1, x => x + 10);
console.log(tab1); ----------> [11, 12, 13, 14, 15]

traiterTableau(tab1, x => x * 10);
console.log(tab1); ----------> [110, 120, 130, 140, 150]
```

fonction avec un paramètre **x** et qui retourne la valeur **x** incrémentée de **10**

Fonction fléchées (Arrow functions)

- Syntaxe générale

```
([parametre1 [, parametre2, [parametre3 … ]]]) => {
        instructions …
}
```

ce qui est entre [et] est facultatif

- Si le corps de la fonction ne contient qu'une seule instruction retournant une valeur les { } peuvent être omises ainsi que l'instruction **return**
  - Ex : `(a, b) => { return a + b }` ⇔ `(a, b) => a + b`

- Si la liste des paramètres ne contient qu'un seul paramètre les ( ) peuvent être omises
  - Ex : `(a) => return a * a` ⇔ `a => a * a`

- S'il n'y a aucun paramètre, cela doit être indiqué par une paire de parenthèses ( )
  - Ex : `() => "Bonjour monde cruel"`

Plan

- Fonctions
  - déclaration
  - valeur de retour
  - invocation
  - variables de type fonction (*first class functions*)
  - fonctions d'ordre supérieur (*high order function*)
  - fonctions anonymes
  - fonction fléchées(*arrow functions*)
- Portée (*Scope*) des variables
  - variable locale, variable globale
  - différence entre `let` et `var`
  - Remontée des variables (hoisting)
- Bonnes pratiques de codage

Portée (*scope*) des variables

- Portée (*scope*) d'une déclaration : la région du programme où l'entité déclarée (variable, objet, fonction)* peut être accédée via son identifiant.
- En JavaScript 3 portées** :
  - **Globale** : un identifiant défini dans la portée globale est accessible dans tout le script où il est déclaré
  - **Locale** : un identifiant défini localement à une fonction est accessible dans tout le corps de la fonction où il est déclaré
  - **Bloc** : un identifiant de variable déclarée avec `let` ou `const` défini dans un bloc (`{ }`) est accessible uniquement dans le bloc (et dans tous les blocs imbriqués) où il est déclaré.

* En JavaScript objets et fonctions sont aussi des variables

** Il y a aussi une portée module (on reviendra dessus lorsque l'on parlera des modules)

Portée (scope) des variables

- variables déclarées via **`const`** et **`let`** ont une portée de bloc (*block-scoped*)
  - accessibles dans le bloc où elles sont déclarées et dans tous les blocs internes à ce bloc.

Exécution avec NodeJS

blockScope.js > ...

```
1   console.log("Debut du script\n");
2   let v1 = 1;
3   console.log(`Global : v1 = ${v1}\n`);
4   { // bloc A
5     const c1 = 1;
6     console.log(`Global : v1 = ${v1}`);
7     console.log(`Bloc A : c1 = ${c1}\n`);
8     { // bloc B
9       let v2 = 2;
10      console.log(`Global : v1 = ${v1}`);
11      console.log(`Bloc A : c1 = ${c1}`);
12      console.log(`Bloc B : v2 = ${v2}\n`);
13      { // bloc C
14        const c2 = 2;
15        console.log(`Global : v1 = ${v1}`);
16        console.log(`Bloc A : c1 = ${c1}`);
17        console.log(`Bloc B : v2 = ${v2}`);
18        console.log(`Bloc C : c2 = ${c2}\n`);
19      } // fin bloc C
20    } // fin bloc B
21  } // fin bloc A
22  console.log(`Global : v1 = ${v1}\n`);
23  console.log(`Bloc A : c1 = ${c1}\n`);
24  console.log("fin 1er script\n");
25  console.log("Fin du script\n");
```

```
PS P:\M2CCI> node .\blockScope.js
Début du script

Global : v1 = 1

Global : v1 = 1
Bloc A : c1 = 1

Global : v1 = 1
Bloc A : c1 = 1
Bloc B : v2 = 2

Global : v1 = 1
Bloc A : c1 = 1
Bloc B : v2 = 2
Bloc C : c2 = 2

Global : v1 = 1

P:\M2CCI\blockScope.js:23
console.log(`Bloc A : c1 = ${c1}\n`);
                                ^


ReferenceError: c1 is not defined
    at Object.<anonymous> (P:\M2CCI\blockScope.js:23:30)
...
Node.js v17.7.2
PS P:_M2CCI>
```

c1 n'est pas connue : erreur d'exécution, les instructions suivantes du script ne sont pas exécutées

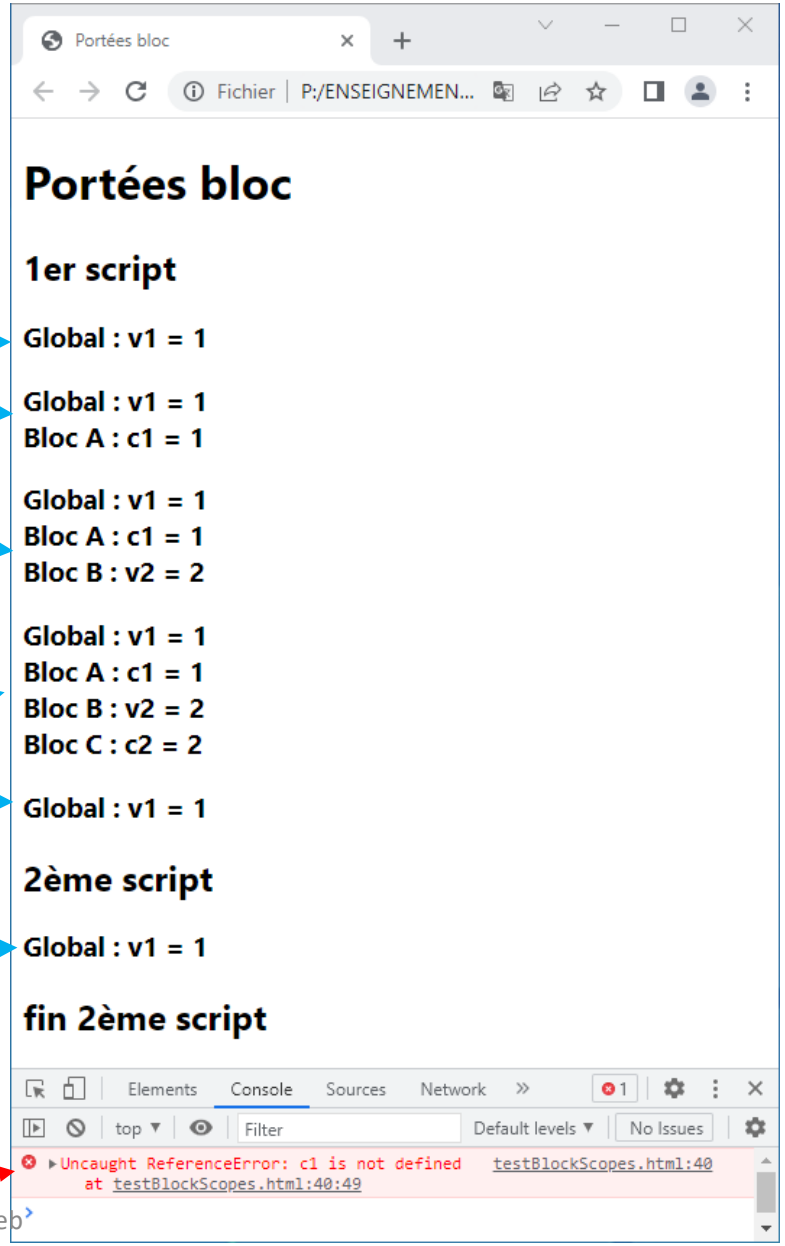
Portée (scope) des variables

Exécution dans un navigateur (Chrome)

```
<!DOCTYPE html>
<html lang="en">
<head>···
</head>
<body>
    <h1>Portées bloc</h1>
    <div>
        <script>
            document.write("<h2>1er script</h2>");
            let v1 = 1;
            document.write(`<h3>Global : v1 = ${v1}</h3>`)
            {  // bloc A
                const c1 = 1;
                document.write(`<h3>Global : v1 = ${v1}<br>\
                    Bloc A : c1 = ${c1}</h3>`);
                { // bloc B
                    let v2 = 2;
                    document.write(`<h3>Global : v1 = ${v1}<br>\
                        Bloc A : c1 = ${c1}<br>\
                        Bloc B : v2 = ${v2}</h3>`);
                    {  // bloc C
                        const c2 = 2;
                        document.write(`<h3>Global : v1 = ${v1}<br>\
                            Bloc A : c1 = ${c1}<br>\
                            Bloc B : v2 = ${v2}<br>\
                            Bloc C : c2 = ${c2}</h3>`);
                    }
                }
            }
            document.write(`<h3>Global : v1 = ${v1}</h3>`);
            document.write(`<h3>Bloc A : c1 = ${c1}</h3>`);
            document.write('<h2>fin 1er script</h2>');
        </script>
        <script>
            document.write("<h2>2ème script</h2>");
            document.write(`<h3>Global : v1 = ${v1}</h3>`);
            document.write('<h2>fin 2ème script</h2>');
        </script>
    </div>
</body>
</html>
```

c1 n'est pas connue : erreur d'exécution, les instructions suivantes du script ne sont pas exécutées

v1 déclarée dans le script 1 est accessible dans le script 2



Fonctions : définitions multiples

- JavaScript n'utilise pas les signatures pour identifier les fonctions
  - pas de surcharge comme en Java
  - si dans une même portée deux fonctions ont le même nom , l'interpréteur JavaScript utilise celle définie en dernier



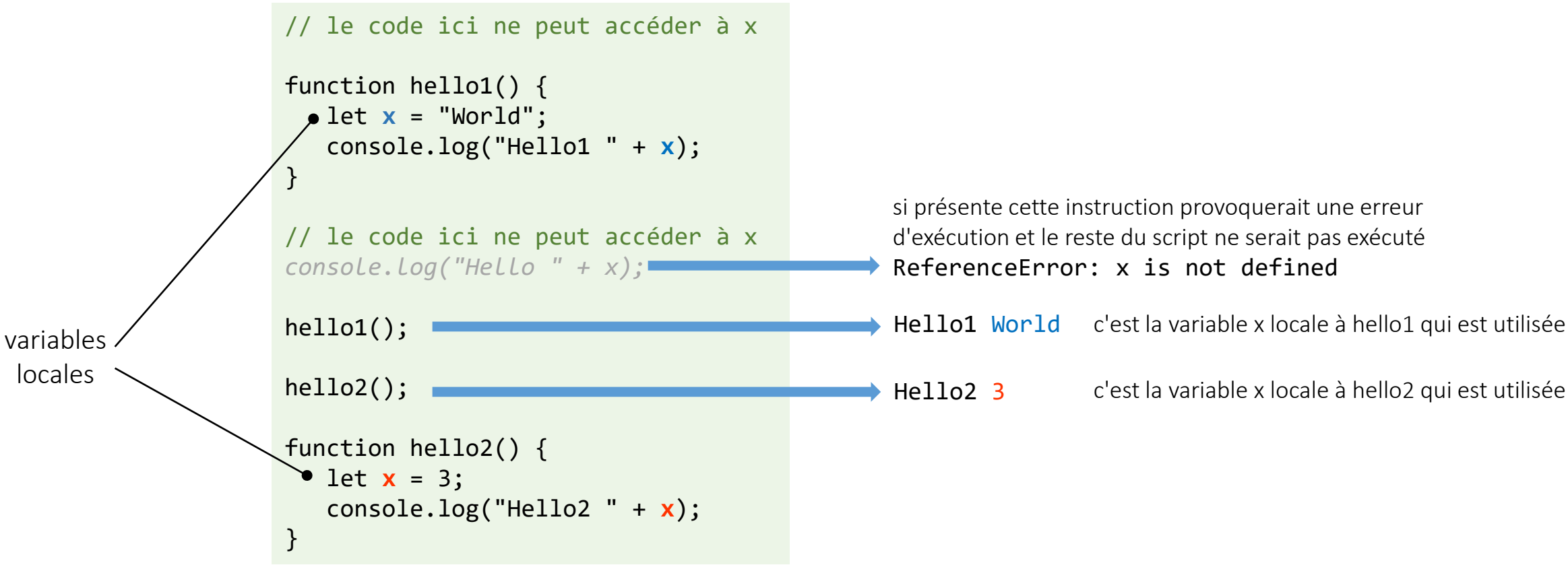
```
function helloWorld(nb) {
    for (let i=0; i < nb; i++) {
       console.log("Hello World !");
    }
}

function helloWorld(nb,message) {
   for (let i=0; i < nb; i++) {
      console.log(message + " World !");
   }
}

helloWorld(2, "Hello");  ----> Hello World !
                               Hello World !
helloWorld(2); ---------------> undefined World !
                               undefined World !
```

Portée (scope) des variables

- une variable déclarée à l'intérieur d'une fonction est une **variable locale** à cette fonction
  - Créée au début de l'exécution de la fonction, détruite à la fin de son exécution
  - Ne peut être accédée qu'à l'intérieur de la fonction
  - Des variables avec le même nom peuvent être utilisées dans des fonctions différentes.



Portée (scope) des variables

- une variable déclarée à l'extérieur d'une fonction et en dehors de tout bloc est une **variable globale**
  - Toutes les fonctions du script peuvent y accéder.

variable globale

```
function hello1() {
    console.log("Hello1 " + x );
}

// x est une variable globale elle
// est accessible partout
let x = "World";

hello1();

hello2();

console.log("x : " +  x);

function hello2() {
    x = 3;
    console.log("Hello2 " + x );
}
```

`Hello1 World` — c'est la variable **x** globale qui est utilisée

`Hello2 3` — c'est la variable **x** globale qui est utilisée et modifiée

`x: 3` — c'est la variable **x** globale qui est utilisée
Elle a été modifiée par **hello2**.

© UGA-2026-2027 Philippe GENOUD - M2 CCI - M2 GEOMAS - Cours Technos Web

Portée (scope) des variables

- une variable déclarée à l'extérieur d'une fonction est une **variable globale**
  - Toutes les fonctions du script peuvent y accéder.
  - Quand une variable locale a le même nom qu'une variable globale, la déclaration locale masque la déclaration globale

variable globale

Variable locale qui masque la variable globale

```
function hello1() {
    console.log("Hello1 " + x );
}

// x est une variable globale elle
// est accessible partout
let x = "World";

hello1();

hello2();

console.log("x : " +  x)

function hello2() {
    let x = 3;
    console.log("Hello2 " + x );
}
```

```
Hello1 World
Hello2 3
x: World
```

c'est la variable **x globale** qui est utilisée

c'est la variable **x locale** qui est utilisée

c'est la variable **x globale** qui est utilisée

Portée (scope) des variables



- si une valeur est affectée à une variable non déclarée, celle-ci devient **automatiquement** une **variable globale**

Variable globale automatique

```
function hello1() {
    x = 3;
    console.log("Hello1 " + x );
}



hello1();

console.log("x : " +  x);

hello2();

console.log("x : " +  x);

function hello2() {
    x = x + 1 ;
    console.log("Hello2 " + x );
    x = 'World' ;
}
```

x n'a pas été déclarée, une variable globale de nom **x** est automatiquement créé

```
Hello1 3

x: 3

Hello2 4

x: World
```

Portée (scope) des variables

- si une variable non déclarée est lue avant d'avoir été initialisée, une erreur d'exécution est provoquée

Variable globale automatique

```
function hello1() {
    x = 3;
    console.log("Hello1 " + x );
}

console.log("x : " +  x) ;

hello1();

console.log("x : " +  x);

hello2();

console.log("x : " +  x);

function hello2() {
    x = x + 1 ;
    console.log("Hello2 " + x );
    x = 'World' ;
}
```

Tentative de lecture alors que la variable n'a pas été initialisée

```
ReferenceError: x is not defined
```

Les instructions suivant l'erreur ne sont pas exécutées

Portée (scope) des variables

- Variable `not defined` ≠ variable `undefined`

```
console.log("x : " +  x) ;
```

x n'a pas été déclarée, erreur d'exécution

ReferenceError: x is not defined

```
let x;
console.log("x : " +  x) ;

let y = 1;
console.log("y : " +  y) ;
```

x déclarée, mais pas initialisée

undefined

y déclarée, et initialisée

1

Portée (scope) des variables

- Que fait ce programme ? Quel résultat est affiché ?

```
1
2
3   function cherche(tab, value) {
4       let i = 0;
5       let trouvé = false;
6
7       while (i <= tab.length && ! trouvé) {
8           if (tab[i] === value) {
9               trouve = true;
10          } else {
11              i++;
12          }
13      }
14      if (trouvé) {
15         console.log('le tableau contient la valeur ' + value +  ' sa position est ' + resultat);
16      } else {
17         console.log('le tableau ne contient pas la valeur ' + value);
18      }
19  }
20
21  cherche([2, 4, 14, 23, 12, 7, 10], 34);
22
23  cherche([2, 4, 14, 23, 12, 7, 10], 14);
24
```


© UGA-2026-2027 Philippe GENOUD - M2 CCI - M2 GEOMAS - Cours Technos Web

Portée (scope) des variables

- Que fait ce programme ? Quel résultat est affiché ?

```
1
2
3   function cherche(tab, value) {
4       let i = 0;
5       let trouvé = false;
6
7       while (i <= tab.length && ! trouvé) {
8           if (tab[i] === value) {
9               trouve = true;
10          } else {
11              i++;
12          }
13      }
14      if (trouvé) {
15         console.log('le tableau contient la valeur ' + value +  ' sa position est ' + resultat);
16      } else {
17         console.log('le tableau ne contient pas la valeur ' + value);
18      }
19  }
20
21  cherche([2, 4, 14, 23, 12, 7, 10], 34);
22
23  cherche([2, 4, 14, 23, 12, 7, 10], 14);
24
```



trouve est une variable globale automatique différente de la variable locale **trouvé**

--> le tableau ne contient pas la valeur 34

Boucle infinie



variables globales automatiques sont **dangereuses**,
elles peuvent conduire à des erreurs silencieuses difficiles à détecter

Avec ES5 utiliser le mode strict tu peux



Mode strict ("use strict")

- introduit avec ES5
  - permet de choisir une variante restrictive de JavaScript
  - facilite l'écriture de code JavaScript plus sûr et plus efficace.
  - change "la mauvaise syntaxe" précédemment tolérée en de véritables erreurs
    - ex : plus la possibilité d'utiliser des variables sans les avoir déclarées

JavaScript «normal»

```
let maVariable = true;

...
while (maVariable) {
   ...
   if (...) {
       maVaraible = false;
   }
   ...
}
...
```



Faute de frappe
→ crée une nouvelle variable globale
→ boucle infinie

JavaScript mode strict

```
"use strict" ;
let maVariable = true;

...
while (maVariable) {
   ...
   if (...) {
       maVaraible = false;
   }
   ...
}
...
```

expression qui doit être au début du code JavaScript ou au début d'une fonction. Ignorée par versions antérieures de JavaScript



Provoque une erreur
ReferenceError: maVaraible is not defined.

pour plus de détails sur le mode strict https://www.w3schools.com/js/js_strict.asp, https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Strict_mode

Portée (scope) des variables

- En JavaScript plusieurs manières de définir des variables

- Variables globales automatiques
- Déclaration explicite avec le mot clé `var`



- Déclaration explicite avec les mot clés `let` ou `const` (ES6+)

```
var unTexte = "Hello World" ;
```

?

```
let unTexte = "Hello World" ;
```

Quelle différence ?

Portée (scope) des variables

```
var x;
var y = 1;

console.log("x : " +  x);
console.log("y : " +  y);
```

x déclarée, mais pas initialisée
undefined
1

```
let x;
let y = 1;

console.log("x : " +  x);
console.log("y : " +  y);
```

x déclarée, mais pas initialisée
undefined
1

```
console.log("x : " +  x);
console.log("y : " +  y);

var x;
var y = 1;
```

undefined
undefined



```
console.log("x : " +  x);
console.log("y : " +  y);

let x;
let y = 1;
```

Provoque une erreur
ReferenceError: can't access lexical declaration `x' before initialization

```
var x, y;

console.log("x : " +  x);
console.log("y : " +  y);

y = 1;
```

- Les variables déclarées avec `var` peuvent être utilisées avant leur déclaration ce qui n'est pas le cas avec `let`.
- JavaScript fait remonter les déclarations (*hoist= hisser*) de variables faites avec `var` au début de la portée (scope) courante (le script ou la fonction courante).

⚠ Seules les déclarations sont remontées, pas les initialisations.

Portée (scope) des variables

```
function hello1 () {
   console.log("Hello1 " + x);
}

var x = "World";

hello1();
hello2();
console.log("x: " + x);

function hello2 () {
   x = 3;
   console.log("Hello2 " + x);
}
```

Hello1 World
Hello2 3
x: 3

```
function hello1 () {
   console.log("Hello1 " + x);
}

hello1();

var x = "World";

hello2();
console.log("x: " + x);

function hello2 () {
   x = 3;
   console.log("Hello2 " + x);
}
```

Hello1 undefined
Hello2 3
x: 3



La variable **x** a été hissée

```
var x;

function hello1 () {
   console.log("Hello1 " + x);
}

hello1();

x = "World";

hello2();
console.log("x: " + x);

function hello2 () {
   x = 3;
   console.log("Hello2 " + x);
}
```

Quand **hello1** est appelée, **x** n'a pas été initialisée

Portée (scope) des variables

```
var x = 1;

function f1() {
   console.log("f1 x: " + x);
}

f1();

console.log("global x: " + x);
```

f1 x: 1

global x: 1

```
var x = 1;

function f1() {
   console.log("f1 x: " + x);
   var x = 10;
}

f1();

console.log("global x: " + x);
```

f1 x: undefined

global x: 1

La variable locale **x** a été hissée

```
var x = 1;

function f1() {
   var x;
   console.log("f1 x: " + x);
   x = 10;
}

f1();

console.log("global x: " + x);
```

La variable locale **x** n'a pas encore été initialisée

Pour en savoir plus voir "Comment le hoisting fonctionne en JavaScript et pourquoi" Fabien Huet, sept. 2014
http://blog.wax-o.com/2014/09/comment-le-hoisting-fonctionne-en-javascript-et-pourquoi/

Portée (scope) des variables

- `var`
  - Déclaration « historique »
  - **Hoisting**
    - Si déclaration à l'intérieur d'une fonction le nom est rangé (hissé) au niveau de la portée de la fonction
    - Si déclaration au niveau global le nom est hissé en tête de la portée globale (attachée à l'objet **`window`** dans le cas d'un navigateur)
    - Possibilité d'utiliser la variable avant son instruction de déclaration
    - Possibilité de déclarer plusieurs fois le même nom dans une même portée, cela ne définit qu'une seule variable.



- `let` ou `const`
  - Nouvelle déclaration ES6+
  - **Pas de hoisting**,
    - la portée d'une variable est limitée au bloc où elle est déclarée
    - Interdiction d'utiliser la variable avant son instruction de déclaration
    - Interdiction de déclarer deux fois le même nom dans un même bloc

`var` *tu oublieras*



Portée (scope) des variables

- `var`

```
ligne: undefined
i: undefined
--------------
1 *****
--------------
i: 6
ligne: 1 *****
```

```
1   console.log("ligne: " + ligne);
2   console.log("i: " + i);
3   console.log("--------------");
4   for (var i = 1; i <= 3; i++) {
5       var ligne = i + " ";
6       for (var i = 0; i < 5; i++) {
7           ligne += "*";
8       }
9       console.log(ligne);
10  }
11  console.log("--------------");
12  console.log("i: " + i);
13  console.log("ligne: " + ligne);
```

```
1   var i;
2   var ligne;
3   console.log("ligne: " + ligne);
4   console.log("i: " + i);
5   console.log("--------------");
6   for (i = 1; i <= 3; i++) {
7       ligne =  i + " ";
8       for (i = 0; i < 5; i++) {
9           ligne += "*";
10      }
11      console.log(ligne);
12  }
13  console.log("--------------");
14  console.log("i: " + i);
15  console.log("ligne: " + ligne);
```

Les variables **`i`** et **`ligne`** ont été hissées dans la portée globale

La variable de contrôle des deux boucles est la même variable: **`i`**

- `let`

```
ReferenceError: ligne is
not defined
```

```
1   console.log("ligne: " + ligne);
2   console.log("i: " + i);
3   console.log("--------------");
4   for (let i = 1; i <= 3; i++) {
5       let ligne = i + " ";
6       for (let i = 0; i < 5; i++) {
7           ligne += "*";
8       }
9       console.log(ligne);
10  }
11  console.log("--------------");
12  console.log("i: " + i);
13  console.log("ligne: " + ligne);
```

Les variables sont dans la portée du bloc où elles sont déclarées
Elles ne sont pas définies en dehors.

`ligne` ne peut être utilisée que depuis sont point de déclaration jusqu'à la fin du bloc où elle est déclarée.

© UGA-2026-2027 Philippe GENOUD - M2 CCI - M2 GEOMAS - Cours Technos Web

Portée (scope) des variables

- `var`

```
ligne: undefined
i: undefined
--------------
1 *****
--------------
i: 6
ligne: 1 *****
```

```
1   console.log("ligne: " + ligne);
2   console.log("i: " + i);
3   console.log("--------------");
4   for (var i = 1; i <= 3; i++) {
5       var ligne = i + " ";
6       for (var i = 0; i < 5; i++) {
7           ligne += "*";
8       }
9       console.log(ligne);
10  }
11  console.log("--------------");
12  console.log("i: " + i);
13  console.log("ligne: " + ligne);
```

```
1   var i;
2   var ligne;
3   console.log("ligne: " + ligne);
4   console.log("i: " + i);
5   console.log("--------------");
6   for (i = 1; i <= 3; i++) {
7       ligne =  i + " ";
8       for (i = 0; i < 5; i++) {
9           ligne += "*";
10      }
11      console.log(ligne);
12  }
13  console.log("--------------");
14  console.log("i: " + i);
15  console.log("ligne: " + ligne);
```

Les variables **`i`** et **`ligne`** ont été hissées dans la portée globale

La variable de contrôle des deux boucles est la même variable: **`i`**

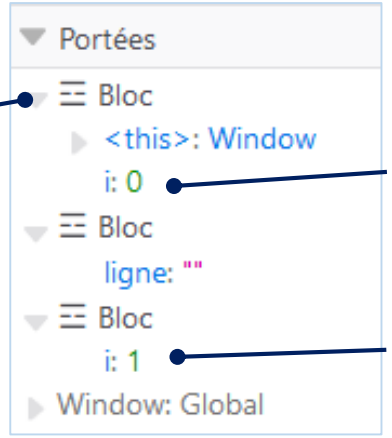
- `let`

```
--------------
1 *****
2 *****
3 *****
--------------
```

```
1   // console.log("ligne: " + ligne);
2   // console.log("i: " + i);
3   console.log("--------------");
4   for (let i = 1; i <= 3; i++) {
5       let ligne = i + " ";
6       for (let i = 0; i < 5; i++) {
7           ligne += "*";
8       }
9       console.log(ligne);
10  }
11  console.log("--------------");
12  // console.log("i: " + i);
13  // console.log("ligne: " + ligne);
```

Lorsque que dans un bloc une variable est déclarée avec le nom d'une variable définie dans un bloc englobant, cela définit une nouvelle variable qui masque la variable dans la portée du bloc englobant.

A un point d'arrêt le debugger affiche la portée du bloc où se trouve l'instruction et les portées englobantes



`i` de la 2ème boucle for

`i` de la 1ère boucle for

© UGA-2026-2027 Philippe GENOUD - M2 CCI - M2 GEOMAS - Cours Technos Web

Bonnes pratiques

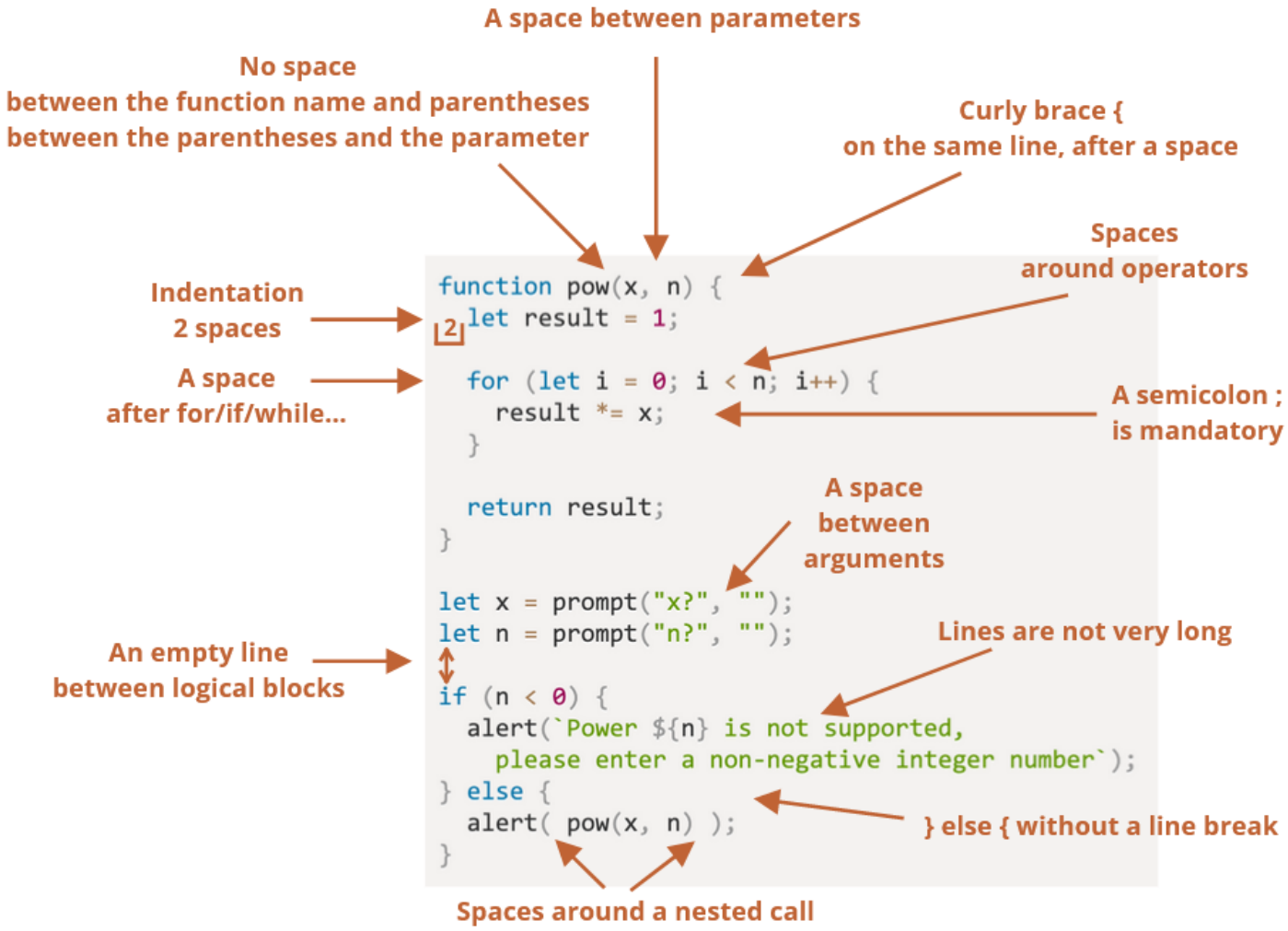


- `let` ou `const` au lieu de `var`
- donner des noms significatifs
- utiliser le camelCase
  - nom d'une variable et d'une fonction commence par une minuscule
- déclarer une seule variable par ligne

- utiliser `===` et `!==` au lieu de `==` et `!=`
- utiliser `const` si la valeur d'une variable ne doit pas changer
  - Éventuellement nommer les constantes avec des identifiants en majuscules

Bonnes pratiques

- Votre code doit être aussi propre et lisible que possible.



© JAVASCRIPT.INFO

https://javascript.info/coding-style

Bonne pratiques

- Commentaires
  - `/* … */`
    sur plusieurs lignes
  - `//`
    sur une ligne
  - `/** */`
    commentaires documentant (JSDoc) pour les fonctions
    - Utilisés par l'éditeur (ex VScode)
    - Utilisés pour générer de la documentation (JSDoc 3) par ex en HTML

```
/*
 * degres.js programme de conversion de températures de ° Fahreinheit vers ° Celsius.
 *
 * utilise la fonction utilitaire encore() (définie dans le module utils)
 * afin que l'utilisateur puisse si il le souhaite effectuer plusieurs conversions
 * successives.
 *
 * utilise le module readline-sync pour les lectures au clavier
 * voir https://www.npmjs.com/package/readline-sync
 */

const readline = require('readline-sync'); // pour utiliser le module readline-sync
const utils = require('./utils.js'); // pour utiliser le module utils

/**
 * convertit une valeur de degrès Fahreinheit vers les degrès Celsius
 * @param {number} tempF la valeur à convertir en degrés Fahreinheit
 * @returns {number} la valeur en degrés Celsius
 */
function fahreinheit2Celsius(tempF) {
    return (5 / 9) * (tempF - 32);
}

//----------------------------
// Le programme principal
//----------------------------
do {
    let tempF = readline.questionFloat("donnez une température en degrés Fahrenheit : ");
    console.log("la température en degrés Celsius est " +
                                    fahreinheit2Celsius(tempF).toFixed(2));
} while (utils.encore("voulez-vous recommencer ? "));
console.log("Au revoir !");
```