JAVASCRIPT

Table des matières

I. Utilisez les données et les types de données dans JavaScript

1. Déclarer des variables et modifier leur valeur

https://javascript.info/variables

Pour déclarer :	Les variables sont mutables : on peut les modifier :	Déclarer plusieurs variables :

`let numberOfCats = 2;`
`let numberOfDogs = 4;`
	`let numberOfCats = 3;`
	`numberOfCats = 4;`
		`let user = 'John',`
		`age = 25,`
		`message = 'Hello';`

Opérateurs arithmétiques – travail sur des nombres – Addition et Soustraction :

`let totalCDs = 67;`
`let totalVinyls = 34;`
`let totalMusic = totalCDs + totalVinyls;`
	`let cookiesInJar = 10;`
	`let cookiesRemoved = 2;`
	`let cookiesLeftInJar = cookiesInJar - cookiesRemoved;`

Les variables :

`let meal = 'Enchiladas';`
`console.log(meal); // Output: Enchiladas`
`meal = 'Burrito';`
`console.log(meal); // Output: Burrito`
	`let price;`
	`console.log(price); // Output: undefined`
	`price = 350;`
	`console.log(price); // Output: 350`

`const entree = 'Enchiladas';`
`console.log(entree); // Output : Enchiladas`
`entree = 'Tacos'; //TypeError : Assignement to constant variable`
	Les constantes ne sont pas mutables. On ne peut pas changer sa valeur :

Les variables qui n'ont pas été initialisées sont de type *undefined*.

Convertir float en int :

```
const int1 = 19.8 | 0;
console.log(int1); // 19
const int2 = 1553 / 10 | 0;
console.log(int2): // 155
```

2. Les nombres

https://www.w3schools.com/js/js_arithmetic.asp https://www.w3schools.com/js/js_numbers.asp
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/Math
https://javascript.info/operators

Opérateurs arithmétiques – travail sur des nombres – Multiplication et Division :

```
let x = 20;
x -= 5; // Can be written as x = x - 5
console.log(x); // Output: 15

let y = 50;
y *= 2; // Can be written as y = y * 2
console.log(y); // Output: 100

let z = 8;
z /= 2; // Can be written as z = z / 2
console.log(z); // Output: 4
```

```
let a = 10;
a++;
console.log(a); // Output: 11

let b = 20;
b--;
console.log(b); // Output: 19
```

```
let costPerProduct = 20;
let numberOfProducts = 5;
let totalCost = costPerProduct *
numberOfProducts;
let averageCostPerProduct = totalCost /
numberOfProducts;
```

Attention aux nombres à virgules :

```
let integerCalculation = 1 + 2;  // donne 3
let weirdCalculation = 0.1 + 0.2;  // on
attend 0.3, réponse réelle
0.30000000000000004
```

- Math.floor, Math.ceil, Math.random

```
console.log(Math.floor(Math.random() * 100)) // Génère un nombre entier entre 0 et 100
console.log(Math.ceil(43.8)) // arrondi à l'excès
```

- Tester si un argument est un entier

```
console.log(Number.isInteger(2017)) //Vérifie si l'argument est un entier
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

3. Les chaînes de caractères : string

https://developer.mozilla.org/fr/docs/Learn/JavaScript/First_steps/Strings
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/String
https://www.w3schools.com/js/js_strings.asp
https://www.w3schools.com/jsref/jsref_obj_string.asp

```
let firstName = "Will";
let lastName = 'Alexander';
```

```
let wholeName = firstName + " " + lastName;
 // valeur: "Will Alexander"
```

Une autre manière de faire de la concaténation : String interpolation :

```
let myName = 'Florent';
let myCity = 'Nice';
console.log(`My name is ${myName}. My favorite city is ${myCity}.`)
// Output: My name is Florent. My favorite city is Nice.
const bigmouth = 'I\'ve got no right to take my place…';
```

- .toString()

```
const myNum2 = 123;
const myString2 = myNum2.toString();
console.log(typeof myString2); //string
```

- .length

```
let txt = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
console.log(txt.length); // 26
```

Extraction de partie de chaîne :

- .slice(start, end)

```
let str = "Apple, Banana, Kiwi";
console.log(str.slice(7, 13)) ; //Banana
console.log(str.slice(-12,-6)); //Banana
console.log(str.slice(7)) ; //Banana, Kiwi
```

- .substring(start, end)

Comme slice() mais si les valeurs sont négatives, elles sont traitées comme 0.

- .substr(start, end)

Similaire à slice() mais le second paramètre est la longueur de la partie à extraire.

```
console.log(str.substr(7, 6)) //Banana
```

Extraction de caractères :

- .chartAt()

Retourne le caractère à la position spécifiée :

```
let text = "HELLO WORLD";
let char = text.charAt(0);
console.log(char); // "H"
```

- .charCodeAt()

Retourne le unicode du caractère indiqué en argument :

```
let char1 = text.charCodeAt(0);
console.log(char1); // 72
```

- Property Access : []

```
let char = text[0];
console.log(char); //"H"
```

Cette méthode fonctionne comme les tableaux alors que les string n'en sont pas. Si aucun caractère n'est trouvé, [] retourne undefined alors que charAt() retourne un string vide.

- .replace(init, new)

Remplace la 1ère occurence

```
let text = "Please visit Microsoft!";
let newText = text.replace("Microsoft", "W3Schools");
console.log(newText); //"Please visit W3Schools!"
```

La méthode .replace() est sensible à la case. Pour contrer cela, il faut passer aux RegEx qui ne sont pas écrites entre guillemets mais entre /. On rajoute un i à la fin

```
let text = "Please visit Microsoft!";
let newText = text.replace(/MICROSOFT/i, "W3Schools");
console.log(newText); //"Please visit W3Schools!"
```

Pour remplacer toutes les occurrences. Même procédé en rajoutant un g à la fin :

```
let text = "Please visit Microsoft and Microsoft!";
let newText = text.replace(/Microsoft/g, "W3Schools");
console.log(newText); //"Please visit W3Schools and W3Schools!"
```

- .replaceAll(pattern, replacement)

```
let str = "I like dogs because dogs are adorable";
str.replaceAll("dogs", "cats"); //"I like cats because cats are adorable
```

- .toUpperCase() et .toLowerCase()

```
let text1 = "Hello World!";
let text2 = text1.toUpperCase();
let text3 = text1.toLowerCase();
```

```
console.log(text2); // HELLO WORLD!
console.log(text3); // hello world!
```

- .concat()

Permet de concaténer plusieurs chaînes. Les deux méthodes suivantes sont identiques :

```
let text1 = "Hello";
let text2 = "World";
let text3 = text1.concat(" ", text2);
```

```
text = "Hello" + " " + "World!";
text = "Hello".concat(" ", "World!");
```

- .trim()

Cette méthode supprime les espaces au début et à la fin d'une chaîne de caractères :

```
let text1 = "      Hello World!      ";
let text2 = text1.trim();
```

```
console.log(text2); //"Hello World!"
```

- .padStart() et .padEnd()

```
let text = "5";
let padded1 = text.padStart(4,"x");
let padded2 = text.padStart(4,"0");
let padded3 = text.padEnd(4,"x");
let padded4 = text.padEnd(4,"0");
```

```
console.log(padded1); //xxx5
console.log(padded2); //0005
console.log(padded3); //5xxx
console.log(padded4); //5000
```

Fonctionne qu'avec les string. Pour le faire fonctionner avec des nombres, il faut les convertir en string avant.

- .split()

Cette méthode convertir les string en array :

```
text.split(",")    // Split on commas
text.split(" ")    // Split on spaces
text.split("|")    // Split on pipe
text.split("")    // Split at each char
```

Si aucun séparateur n'est spécifié, le tableau retourné contiendra toute la chaîne à l'indice 0.

- .startsWith, .endsWith

```
console.log('Hey'.startsWith('He')); //Affiche true
console.log('Hey'.endsWith('ey')); //Affiche true
```

Ces méthodes prennent une deuxième valeur en argument qui change l'index de début (0 par défaut) pour *startsWidth()* et de fin pour *endsWith()*. Méthode sensible à la casse.

- .indexOf(), .lastIndexOf()

```
let str = "Please locate where 'locate' occurs!";
str.indexOf("locate"); // 7
```

```
let str = "Please locate where 'locate' occurs!";
str.lastIndexOf("locate"); // 21
```

Retourne -1 si absent

```
let str = "Please locate where 'locate' occurs!";
str.indexOf("locate", 15); // 21
```

```
let str = "Please locate where 'locate' occurs!";
str.lastIndexOf("locate", 15); //7
```

Les deux méthodes acceptent un deuxième argument qui est la position de début de recherche (sens droite gauche)

- .search()

```
let str = "Please locate where 'locate' occurs!";
str.search("locate"); // 7
```

La différence entre *search()* et *indexOf()* est que *search()* ne peut pas prendre de second argument et que *indexOf()* ne peut pas prendre de puissantes valeurs de recherche comme les regEx.

- .match()

```
let text = "The rain in SPAIN stays mainly in the plain";
console.log(text.match(/ain/g)); // ["ain", "ain", "ain"]
```

Permet de chercher des expressions régulières dans un string et en retourne la liste. Si on enlève le g cela retourne le premier. Si rien est trouvé *null* est renvoyé.

```
let text = "The rain in SPAIN stays mainly in the plain";
console.log(text.match(/ain/gi)); //["ain", "AIN", "ain", "ain"]
```

- .includes()

Cette méthode retourne *true* si le string contient la valeur spécifiée ; *false* sinon.

```
let text1 = "Hello world, welcome to the universe.";
console.log(text1.includes("world")); // true
```

Accepte un 2e argument pour le début de la position de recherche

```
let text2 = "Hello world, welcome to the universe.";
console.log(text2.includes("world", 12)); // false
```

- Conversion nombre <-> string

```
let a = 10;
let b = "20";
console.log(a + b); // "1020"
console.log(a + +b); //30
```

```
let a = 10;
console.log(10 + ""); //"10"
```

4. Type boolean (valeur logique)

```
let userIsSignedIn = true;
let userIsAdmin = false;
```

JavaScript est un langage à types dynamiques et à typage faible : on peut initialiser une variable en temps que nombre et la réaffecter comme une chaîne ou tout autre type. Attention à faire cela avec précaution pour éviter tout comportement inattendu.

Il existe trois autres types de données primitifs en JavaScript : null, undefined et symbol.

5. Connaître le type : opérateur typeof

```
const unknown1 = 'foo';
console.log(typeof unknown1); // Output: string
const unknown2 = 10;
console.log(typeof unknown2); // Output: number
const unknown3 = true;
console.log(typeof unknown3); // Output: boolean
```

II. Les tableaux

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
https://www.w3schools.com/js/js_arrays.asp
https://www.w3schools.com/jsref/jsref_obj_array.asp

1. Introduction aux tableaux

Déclaration d'un tableau :

```
const cars = ["Saab", "Volvo", "BMW"];
const cars = new Array("Saab", "Volvo", "BMW");
```

Tableau Vide :

```
const cars = new Array();
const cars = [];
```

Attention pour les tableaux à un élément :

```
// Create an array with one element:
const points = [40];
// Create an array with 40 undefined elements:
const points = new Array(40);
```

Accéder aux éléments :

```
const groceryList = ['bread', 'tomatoes', 'milk'];
groceryList[0] // le 1er terme groceryList[2]; groceryList.at(2); // le 3e terme
groceryList.at(-1); // le dernier  groceryList.at(-2); // l'avant dernier
groceryList[5]; // undefined
```

On a défini *groceryList* à l'aide de *const*, on ne peut donc pas assigner un nouveau tableau à *groceryList* mais on peut modifier le tableau existant.

Accéder au dernier élément :

```
let car = cars[cars.length - 1]
```

Modifier un élément :

```
cars[0] = "Opel";
```

Les éléments des tableaux peuvent eux-mêmes être également des tableaux ou aussi des objets. Les tableaux ont numéros comme index alors que les objets ont des noms comme index. Si on utilise des noms comme index, JavaScript va redéfinir le tableau en objet. Les tableaux sont une sorte spéciale d'objets avec des index numérotés.

```
console.log(typeof cars); // Output: objet
```

Pour vérifier que l'on est bien en présence d'un tableau :

```
Array.isArray(cars); // true
```

```
cars instanceof Array; // true
```

Les tableaux peuvent être imbriqués :

```
let numberClusters = [[1,2], [3, 4], [5, 6]];
const target = numberClusters[2][1];
console.log(target); //Output 6
```

Et désinbriqués :

```
const arr1 = [0, 1, 2, [3, 4]];
console.log(arr1.flat());
// expected output: [0, 1, 2, 3, 4]
```

2. Les méthodes sur les tableaux

- Array.from()

```
console.log(Array.from("ABCDEFG")); //["A", "B", "C", "D", "E", "F", "G"]
```

La méthode *Array.from()* retourne un tableau avec n'importe quel objet qui a une propriété *length* ou n'importe quel objet itérable.

Méthode sur les tableaux :

```
cars.length   // Returns the number of elements
cars.sort()   // Sorts the array
```

Remplir un tableau : .fill()

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.fill("Kiwi", 2, 4);
```

Permet de remplir un tableau avec la même valeur. Les 2e et 3e arguments sont facultatifs. Ils renseignent l'indice du début et de fin du remplissage.

Ajouter des éléments dans un tableau :

```
const fruits = ["Banana", "Orange", "Apple"];
```

- .push(*args)

```
let len = fruits.push("Lemon");
console.log(len); // Output 4
// Adds a new element (Lemon) to fruits
fruits.push("Lemon", "pineapple");
```

```
fruits[fruits.length] = "Lemon";
// Adds "Lemon" to fruits
```

Attention à cette méthode :

```
fruits[6] = "Lemon";
// Creates undefined "holes" in fruits
```

- .unshift(arg)

```
// Pour rajouter un terme en premier
groceryList.unshift('popcorn');
console.log(groceryList);
//Output [ 'popcorn', bread, 'tomato', milk']
```

Les méthode .push() et .unshift() retourne la longueur du nouveau tableau et modifient le tableau initial.

- .splice()

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 0, "Lemon", "Kiwi"); // ["Banana", "Orange", "Lemon", "Kiwi", "Apple",
"Mango"]
fruits.splice(2, 1, "Lemon", "Kiwi"); //["Banana", "Orange", "Lemon", "Kiwi", "Mango"]
```

Le 1er paramètre indique là où on va ajouter les éléments.
Le 2e indique combien d'éléments vont être supprimés.
Les autres paramètres indique les éléments qui vont être ajoutés.

- .fill()

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.fill("Kiwi", 2, 4);
```

Permet de remplir un tableau avec la même valeur. Les 2e et 3e arguments sont facultatifs. Ils renseignent l'indice du début et de fin du remplissage.

```
// Crée un tableau de 6 éléments string vide
const output = Array(6).fill('');
// Crée une matrice 4 tableaux de 3 éléments
const matrix = Array(4).fill(0).map( () => Array(3).fill(0));
```

Supprimer des éléments dans un tableau

- .shift()

```
const groceryList = ['orange juice', 'bananas', 'coffee beans', 'brown rice', 'pasta',
'coconut oil', 'plantains'];
//Pour supprimer le 1er terme
groceryList.shift();
console.log(groceryList);
//Output : [ 'bananas', 'coffee beans', 'brown rice', 'pasta', 'coconut oil', plantains' ]
```

- .pop()

```
const groceryList = ['bread', 'tomatoes', 'milk', 'apple', 'eggs'];
// Pour supprimer le dernier élément du tableau
const removed = groceryList.pop()
console.log(groceryList)
// Output [ 'bread', 'tomatoes', 'milk', 'apple' ]
console.log(removed)
// Output eggs
```

Les méthode .pop() et .shift() retournent la valeur du tableau qui a été supprimée et modifie également le tableau initial.

- .splice()

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.splice(2, 1);
console.log(fruits); //["Banana", "Orange", "Mango"]
```

- delete

Attention, cette méthode laisse une valeur *undefined* à la place de l'élément supprimé

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
delete fruits[0];
console.log(fruits); // Output [undefined, "Orange", "Apple", "Mango"]
```

- Vider une liste

```
let array = [12, 555, 66, 3];
array.length = 0; // []
```

Merger/Concaténer des tableaux :

- .concat(*args)

```
const myGirls = ["Cecilie", "Lone"];
const myBoys = ["Emil", "Tobias", "Linus"];
const myChildren = myGirls.concat(myBoys);
console.log(myChildren); // ["Cecilie", "Lone", "Emil", "Tobias", "Linus"]
```

la méthode .concat() ne change pas les tableaux existant, elle renvoie un nouveau tableau. Elle peut prendre en argument plusieurs tableaux séparés par des virgules, mais aussi plusieurs string séparés par des virgules.

Extraire une partie d'un tableau :

- .slice()

```
const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];
const citrus = fruits.slice(3);
console.log(citrus); // ["Apple", "Mango"]
```

La méthode .slice() crée un nouveau tableau et ne modifie pas le tableau d'origine.
Si on met une valeur en argument, le slice retournera la tableau de cet index à la fin.

```
const groceryList = ['orange juice', 'bananas', 'coffee beans', 'brown rice', 'pasta',
'coconut oil', 'plantains'];
//Pour sélectionner une tranche du tableau
```

```
let extract = groceryList.slice(1,4); //Dernier index non inclus
console.log(extract); //Output [ 'bananas', 'coffee beans', 'brown rice' ]
```

Si on met deux arguments, cela correspond à l'indice du début (inclus) jusqu'à l'indice de fin (non inclus).

```
//Tronquer un tableau
let array = [0, 1, 2, 3, 4, 5];
array.length = 3; //[0, 1, 2]
```

Convertir un tableau en string :

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
```

- .toString()

```
fruits.toString();
// Banana,Orange,Apple,Mango
```

- .join(sep)

```
fruits.join(" * ");
// Banana * Orange * Apple * Mango
```

Avec la méthode .join(), on peut choisir le séparateur.

Trier un tableau :

- .sort()

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.sort();
console.log(fruits); //["Apple", "Banana", "Mango", "Orange"]
```

Modifie la liste et la retourne par ordre alphabétique. Attention avec les chiffre : 45 < 5 !

- Pour les chiffres :

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
console.log(points);
//[1, 5, 10, 25, 40, 100]
```

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return b - a});
console.log(points);
//[100, 40, 25, 10, 5, 1]
```

On peut ensuite accéder à la plus grande ou à la plus petite valeur grâce à l'index 0.

- Mélanger un tableau dans le désordre :

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(){return 0.5 - Math.random()});
console.log(points); //[40, 1, 25, 100, 5, 10]
```

- Trier un objet : nombre

```
const cars = [
    {type:"Volvo", year:2016},
    {type:"Saab", year:2001},
    {type:"BMW", year:2010}
  ];
```

```
cars.sort(function(a, b){return a.year - b.year});
// Output [[object Object] {
  type: "Saab",
  year: 2001
}, [object Object] {
  type: "BMW",
  year: 2010
}, [object Object] {
  type: "Volvo",
  year: 2016
}]
```

- Trier un objet : les string:

```
cars.sort(function(a, b){
    let x = a.type.toLowerCase();
    let y = b.type.toLowerCase();
    if (x < y) {return -1;}
    if (x > y) {return 1;}
    return 0;
  });
```

Valeur maximale et minimale dans un tableau :

```
function myArrayMax(arr) {
    return Math.max.apply(null, arr);
  }
console.log(myArrayMax(points));
// Output 100
```

```
const points = [40, 100, 1, 5, 25, 10];
function myArrayMin(arr) {
  return Math.min.apply(null, arr);
}
console.log(myArrayMin(points)); // Output
```

`Math.max.apply(null, [1, 2, 3])` is equivalent to `Math.max(1, 2, 3)`

`Math.min.apply(null, [1, 2, 3])` is equivalent to `Math.min(1, 2, 3)`

```
function myArrayMax(arr) {
    let len = arr.length;
    let max = -Infinity;
    while (len--) {
      if (arr[len] > max) {
        max = arr[len];
      }
    }
    return max;
  }
```

```
function myArrayMin(arr) {
    let len = arr.length;
    let min = Infinity;
    while (len--) {
      if (arr[len] < min) {
        min = arr[len];
      }
    }
    return min;
  }
```

Inverser un tableau :

- .reverse()

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.reverse();
console.log(fruits); //["Mango", "Apple", "Orange", "Banana"]
```

Obtenir l'index d'un élément :

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/indexOf

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
// Pour retourner l'index d'un élément
const appleIndex = fruits.indexOf('Apple');
console.log(appleIndex); // Output 2
```

```
const list = ['apple', 'orange', 'potato', 'tomato', 'potato']
list.lastIndexOf('potato'); //4
list.lastIndexOf('potato', 1); //-1
list.lastIndexOf('potato', 2); //2
list.lastIndexOf('potato', 4); //4
```

Le deuxième argument permet de sélectionner une partie du tableau entre le début du tableau et cet index.

Vérifier la présence d'un élément :

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.includes("Mango");
```

Retourne *true* si l'élément en argument est dans le tableau *fruits.*

- .copyWithin()

Permet de recopier une partie de la liste à l'intérieur d'elle-même :

```
const fruits = ["Banana", "Orange", "Apple", "Mango", "Kiwi", "Papaya"];
fruits.copyWithin(2,0); //Banana,Orange,Banana,Orange,Apple,Mango
fruits.copyWithin(2,1,3); //Banana,Orange,Orange,Apple,Kiwi,Papaya
```

1ère valeur : index à partir de laquelle seront copiés les éléments (required) ;

2e : index de départ source copie défaut 0 ; 3e : index de fin source défaut array length

3. Valeurs et références

```
let numberOfGuests = 20;
let totalNumberOfGuests = numberOfGuests; // 20
```

La valeur 20 est copiée dans la nouvelle variable mais aucun lien n'est maintenu entre les deux variables.

Ce n'est pas le cas avec les objets et tableaux qui sont passés par référence :

```
let artistProfile = {
    name:"Tao Perkington",
    age:27,
    available: true
};

console.log(artistProfile)

let allProfiles = artistProfile;

console.log(allProfiles)

artistProfile.available = false;

console.log(artistProfile)

console.log(allProfiles)
```

```
//Output 1 {
  age: 27,
  available: true,
  name: "Tao Perkington"
}
//Output 2 {
  age: 27,
  available: true,
  name: "Tao Perkington"
}
//Output 3 {
  age: 27,
  available: false,
  name: "Tao Perkington"
}
//Output 4 {
  age: 27,
  available: false,
  name: "Tao Perkington"
}
```

Si on modifie l'objet *artistProfile* après l'avoir copiée dans *allProfiles*, on modifie également *allProfile* car ils font référence au même objet.

4. L'opérateur de propagation spread en JavaScript

```
const a = [1, 2, 3];
```

```
console.log([...a, 4, 5, 6]);
// [1, 2, 3, 4, 5, 6]
```

```
console.log([a, 4, 5, 6]);
// [[1, 2, 3], 4, 5, 6]
```

- Il permet de faire une copie de liste :

```
const copie = [...a];
```

- On peut l'utiliser avec les string :

```
const mot = 'bonjour';
const arrayized = [...mot];
console.log(arrayized); //["b", "o", "n", "j", "o", "u", "r"]
```

- Il permet d'utiliser un tableau comme argument de fonction :

```
function sum(x, y, z) {
    return x + y + z;
}
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // expected output: 6
console.log(sum.apply(null, numbers)); // expected output: 6
```

- Extraction de valeurs avec les listes :

```
const numbers = [1, 2, 3, 4, 5]
const [first, second, ...others] = numbers
console.log(numbers);
```

```
console.log(first); // 1
console.log(second); // 2
console.log(others); // [3, 4, 5]
```

- On peut l'utiliser également avec les objets
    - Copie

```
const newObj = { ...oldObj }
```

    - Création

```
const items = { first, second, ...others }
items //{ first: 1, second: 2, third: 3,
fourth: 4, fifth: 5 }
```

```
const object1 = {name: 'Flavio'}
const object2 = {age: 35}
const object3 = {…object1, …object2 }
```

    - Extraction

```
const { first, second, ...others } = {
    first: 1,
    second: 2,
    third: 3,
    fourth: 4,
    fifth: 5
}
first // 1
second // 2
others // { third: 3, fourth: 4, fifth: 5 }
```

- Nettoyer un tableau des doublons

```
const myArr = ['dogs', 'cats', 1, 1, 'movies', 'movies'];
const uniqueArr = [...new Set(myArr)]; //["dogs", "cats", 1, "movies"]
```

- Maximum et Minmum tableau

```
const numbers = [1, 10, 4, 5, 7];
console.log(Math.max(...numbers)); // 10
console.log(Math.min(...numbers)); // 1
```

III. Les objets

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object#Methods

1. Introduction

Construire un objet :

Un objet vide :

```
let user = new Object(); // syntaxe "constructeur d'objet"
let user = {};  // syntaxe "littéral objet"
```

Les objets en JS sont écrits en JSON (JavaScript Object Notation). Ce sont des séries de paires clés-valeurs séparées par des virgules entre des accolades. Les objets peuvent être enregistrés dans une variable :

```
let spaceship = {
    'Fuel Type': 'Turbo Fuel',
    'Active Duty': true,
    homePlanet: 'Earth',
    color: 'silver',
    numCrew: 5
};
```

Chaque clé est une chaîne. Et les valeurs associées peuvent être de tout type.

Construire des objets permet de regrouper les attributs d'une chose unique à un même emplacement comme un livre, un profil utilisateur, la configuration d'une application. En général on déclare les objets avec le mot clé "const".

Accéder aux données d'un objet :

- Avec la notation pointée : *dot notation*

```
spaceship.homePlanet; // Returns 'Earth',
spaceship.color; // Returns 'silver',
```

- Avec la notation bracket

```
spaceship["homePlanet"]; // Returns 'Earth'
spaceship["color"]; // Returns 'silver'
```

Ou encore :

```
let propertyToAccess = "color";
spaceship[propertyToAccess];
// Returns 'silver'
```

L'intérêt est que l'on met entre bracket une variable qui va contenir le nom de la propriété que l'on souhaite atteindre.

```
let user = {
    name: "John",
    age: 30
};
let key = prompt("What do you want to know about the user?", "name");
alert( user[key] ); // John (si entré "name")
```

Attention, la *dot notation* ne fonctionne pas tout le temps. On doit utiliser *bracket notation* quand les clés ont des nombres, des espaces, des caractères spéciaux.

```
spaceship['Active Duty'];   // Returns true
spaceship['Fuel Type'];   // Returns  'Turbo Fuel'
spaceship['numCrew'];   // Returns 5
spaceship['!!!!!!!!!!!!!!!'];   // Returns undefined
```

Si on veut travailler avec des fonctions, il faut aussi utiliser la *bracket notation* :

```
let returnAnyProp = (objectName, propName) => objectName[propName];
returnAnyProp(spaceship, 'homePlanet'); // Returns 'Earth'
```

Autre méthode pour accéder aux propriétés d'un objet :

```
const home = spaceship.homePlanet;
console.log(home); // Output Earth
```

On peut enregistrer la valeur de la propriété sous n'importe quel nom de variable.

```
const { homePlanet } = spaceship;
console.log(homePlanet); // Output Earth
```

Pour cette deuxième méthode, il est indispensable de mettre entre les accolades, le nom exact de la propriété de l'objet.

Assigner des propriétés

```
spaceship.type = 'alien'; // Changes the value of the type property
spaceship.speed = 'Mach 5'; // Creates a new key of 'speed' with a value of 'Mach 5'
```

Supprimer des propriétés

```
delete spaceship.speed;  // Removes the speed property
```

Objets imbriqués

```
let spaceship = {
    passengers: null,
    telescope: {
      yearBuilt: 2018,
      model: "91031-XLT",
      focalLength: 2032
    },
    crew: {
      captain: {
        name: 'Sandra',
        degree: 'Computer Engineering',
        encourageTeam() { console.log('We got this!') },
       'favorite foods': ['cookies', 'cakes', 'candy', 'spinach'] }
    },
    engine: {
      model: "Nimbus2000"
    }
};
let capFave = spaceship.crew.captain['favorite foods'][0];
console.log(capFave); // Output cookies
```

2. Les méthodes sur les objets

- .keys(), .values() & .entries()

```
const myInfo = {
  name: "Dunsin",
  age: 18,
  color: "blue",
};
```

.keys() retourne un tableau avec les clés :

```
console.log(Object.keys(myInfo));
//["name", "age", "color"]
```

.values() retourne un tableau avec les valeurs

```
console.log(Object.values(myInfo));
//["Dunsin", 18, "blue"]
```

.entries() retourne un tableau de tableaux [clé, valeurs] imbriqués

```
console.log(Object.entries(myInfo));
//[["name", "Dunsin"], ["age", 18],
["color", "blue"]]
```

- .assign permet de fusionner deux objets

```
const other = {
  isReal: true
};
console.log(Object.assign(myInfo, other));
```

```
/* Output {
  age: 18,
  color: "blue",
  isReal: true,
  name: "Dunsin"
} */
```

- .fromEntries()

Inverse de .entries(). Permet de créer un objet à partir d'un tableau :

```
const keyValuePair = [ ['one', 1], ['two', 2] ];
const store = Object.fromEntries(keyValuePair);
console.log(store); /* { one: 1,  two: 2}*/
```

- .freeze() permet de se prévenir d'une modification d'un objet

```
Object.freeze(myInfo);
myInfo.age = 19;
console.log(myInfo);
```

```
/*Output {
  age: 18,
  color: "blue",
  name: "Dunsin"
}*/
```

- .seal() est similaire à .freeze() sauf que l'on peut modifier les valeurs des entrées mais on ne peut pas ajouter ou supprimer des entrées
- .preventExtensions() permet de modifier les valeurs des entrées, supprimer des entrées mais pas en ajouter
- .call(), bind() et apply()

```
let myObject = {
    greet: "hello",
    greeting: function(person){
        console.log(this.greet + person);
    }
};
myObject.greeting("Vaishu");
```

```
let otherObject = {
    greet: "Namaste",
    greeting: function(person){
        console.log(this.greet + person);
    }
};
otherObject.greeting("Vee");
```

Cela fait donc un objet et une méthode à écrire pour chaque objet. Avec la fonction call() On peut donc écrire une méthode et pour l'appliquer à tous les objets en gardant le bénéfice de *this* :

```
let myObject = {
    greet: "hello",
};
let otherObject = {
    greet: "Namaste",
};
```

```
//Fonction pour chaque objet
function greeting(person) {
    console.log(this.greet + person);
}
//Attacher la fonction aux différents objets
greeting.call(otherObject, "Vee");
//NamasteVee
greeting.call(myObject, "Vee"); //helloVee
```

Méthode apply() permet d'accepter les arguments sous forme de liste quand il y en a plus d'un :

```
let obj = {num : 2};
function add(a, b) {
  return this.num + a + b;
}
console.log(add.call(obj, 3, 5)); // 10
console.log(add.apply(obj, [3, 5])); // 10
```

Méthode bind() retourne une fonction qui peut exécutée la méthode plus tard :

```
const func = add.bind(obj, 3, 5);
console.log(func()); // 10
```

Bind() permet aussi d'emprunter une méthode d'un objet différent :

```
let runner = {
  name: 'Runner',
  run: function(speed) {
    console.log(this.name + ' runs at ' +
speed + ' mph.');
  }
};
```

```
let flyer = {
  name: 'Flyer',
  run: function(speed) {
    console.log(this.name + ' flies at ' +
speed + ' mph.');
  }
};
```

```
let fct = runner.run.bind(flyer, 20);
fct(); //"Flyer runs at 20 mph."
```

3. Les propriétés calculées

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {
  [fruit]: 5, // le nom de la propriété est tiré de la variable fruit
};
alert( bag.apple ); // 5 si fruit="apple"
```

Cela signifie que le nom de la propriété doit être extrait de *fruit*. Si un visiteur entre *"apple"*, *bag* deviendra *{apple:5}*. Cela fonctionne comme cela :

```
let fruit = prompt("Which fruit to buy?", "apple");
let bag = {};
// prendre le nom de la propriété de la variable fruit
bag[fruit] = 5;
```

Mais la 1ere méthode a une meilleure apparence.

On peut aussi mettre des expressions plus complexes entre les crochets :

```
let fruit = 'apple';
let bag = {
  [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

4. Valeur de propriété abrégée

Dans le code, on utilise souvent des variables existantes en tant que valeurs pour les noms de propriétés. Donc il y a une propriété abrégée (*shorthand*) pour la rendre plus courte. Ainsi les deux écritures suivantes sont équivalentes :

```
function makeUser(name, age) {
    return {
      name: name,
      age: age,
      // ...autres propriétés
    };
}
let user = makeUser("John", 30);
alert(user.name); // John
```

```
function makeUser(name, age) {
    return {
      name, // pareil que name: name
      age,  // pareil que age: age
      // ...
    };
  }
```

NB : On peut mélanger les deux écritures.

5. Test d'existence d'une propriété avec l'opérateur *in*

Il n'y a pas d'erreur en JS si on veut accéder à une propriété qui n'existe pas. Elle renvoie simplement *undefined*:

```
let user = {};
alert( user.noSuchProperty === undefined ); // true signifie "pas une telle propriété"
```

Il existe également l'opérateur *in* qui fonctionne comme cela :

```
let user = { name: "John", age: 30 };
alert( "age" in user ); // true, user.age existe
alert( "blabla" in user ); // false, user.blabla n'existe pas
```

Le nom de la propriété doit être entre guillemet. Sinon cela signifie que c'est une variable qui doit alors contenir le nom de la propriété à tester :

```
let user = { age: 30 };
let key = "age";
alert( key in user ); // true, la propriété "age" existe
```

Dans la plupart des cas, la comparaison avec *undefined* fonctionne convenable. Mais dans un cas particulier il échoue alors que *in* fonctionnement correctement : C'est lorsque la propriété existe et qu'elle stocke *undefined* :

```
let obj = {
    test: undefined
};
alert( obj.test ); // c'est indéfini, donc - pas une telle propriété ?
alert( "test" in obj ); // true, la propriété existe !
```

6. Boucler dans un objet *for...in*

```
let user = {
    name: "John",
    age: 30,
    isAdmin: true
};
```

```
for(let key in user) {
    // keys
    alert( key );  // name, age, isAdmin
    // valeurs pour les clés
    alert( user[key] ); // John, 30, true
}
```

7. Méthodes dans les objets

Les propriétés sont ce que possède un objet, les méthodes sont ce que fait l'objet.

```
let retreatMessage = 'We no longer wish to conquer your planet.';
const alienShip = {
  retreat () {
    console.log(retreatMessage);
  },
  takeOff() {
    console.log('Spim... Borp... Glix... Blastoff!');
  }
};
alienShip.retreat(); //Output We no longer wish to conquer your planet.
alienShip.takeOff(); // Output Spim... Borp... Glix... Blastoff!
```

8. Les objets sont passés par référence

Comme les tableaux, les objets sont passés par référence. On peut modifier les propriétés d'un objet de manière permanente à l'aide des fonctions :

```
const spaceship = {
    homePlanet : 'Earth',
    color : 'silver'
};
let paintIt = obj => {
    obj.color = 'glorious gold'
};
paintIt(spaceship);
spaceship.color // Returns 'glorious gold'
```

9. Le mot-clé *this*

Le mot clé *this*. Il permet d'accès aux propriétés à l'intérieur de l'objet

```
const robot = {
    model : '1E78V2',
    energyLevel : 100,
    provideInfo() {
      return `I am ${this.model} and my current energy level is ${this.energyLevel}`
    }
};
console.log(robot.provideInfo()); //Output I am 1E78V2 and my current energy level is 100
```

Manipulez les classes

Une classe est un modèle pour un objet dans le code. Elle permet de construire plusieurs objets du même type (appelés instances de la classe).

1. On utilise le mot clé class
2. On utilise ensuite la fonction constructor pour indiquer que chaque *Book* ait un titre, un auteur et un nombre de page.
3. Des instructions à l'intérieur du *constructor* pour attribuer le titre, l'auteur et le nombre de page à l'aide du mot *this*.

```
class Book {
    constructor(title, author, pages) {
        this.title = title;
        this.author = author;
        this.pages = pages;
    }
}
```

Le mot clé *this* fait référence à la nouvelle instance.
Donc on utilise la notation dot pour attribuer les valeurs reçues aux clés correspondantes.

Une fois que la classe est terminée, on peut créer des instances par le mot clé *new*. On peut donc créer facilement et rapidement de nouveaux objets *Book* :

```
let myBook = new Book("L'Histoire de Tao", "Will Alexander", 250);
//Cette ligne crée l'objet suivant :
{
    title: "L'Histoire de Tao",
    author: "Will Alexander",
    pages: 250
}
```

IV. Les constructeurs d'objet

1. Syntaxe

```
function Player(name, marker) {
    this.name = name
    this.marker = marker
    this.sayName = function() {
      console.log(name)
    }
}
```

```
const player = new Player('steve', 'X');

player.sayName() // 'steve'
console.log(player.marker) // 'X'

console.log(player); //  {name: "steve", marker: "X", sayName...}
```

Attention :Il ne faut pas oublier d'utiliser le mot new lors de la construction de l'objet.

La 1ère ligne peut également être : `const Player = function(name, marker) {`

2. Le prototype

Chaque fonction JavaScript a une propriété prototype (vide au début) à laquelle on va attacher des propriétés et des méthodes. Elle principalement utiliser pour l'héritage. Toutes les méthodes et propriétés sur la propriété prototype d'une fonction seront disponibles pour les instances de cette fonction.

Tout objet créé :

- de manière littérale : let obj = {} ou let list = []
- ou à l'aide d'un constructeur let obj = new Object() ; let list = new Array() ; let any = new Anything()

va hériter de son constructeur. Le prototype de la nouvelle instance va hériter du prototype du constructeur, donc de ces méthodes et propriétés. Ex : un objet créé avec new Array, aura Array.prototype comme prototype.

Lorsque l'on utilise des constructeurs pour créer des objets, il est préférable de définir les fonctions sur le prototype de cet objet. Une seule instance de chaque fonction sera alors partagée entre tous les objets crées. Si on déclare la fonction dans le constructeur (comme précédemment) cette fonction sera dupliquée dans tous les objets créé. Ex :

```
function Student(name, grade) {
    this.name = name
    this.grade = grade
  }
```

```
  Student.prototype.sayName = function() {
    console.log(this.name)
  }
  Student.prototype.goToProm = function() {
    console.log("Eh.. go to prom?")
  }
```

L'héritage de prototype est très important. Il permet de faire hériter à un objet, les propriétés. Il existe plusieurs façons pour le faire. Voici la manière recommandée :

On crée un constructeur et on lui ajoute une fonction via son prototype, ainsi qu'un 2e constructeur :

```
function Student() {
}

Student.prototype.sayName = function() {
  console.log(this.name)
}
function EighthGrader(name) {
  this.name = name
  this.grade = 8
}
```

On va greffer le prototype du constructeur Student sur celui du constructeur EightGrader :

```
EighthGrader.prototype =
Object.create(Student.prototype)

const carl = new EighthGrader("carl")
carl.sayName() // console.logs "carl"
carl.grade // 8
```

V. Les fonctions d'usine (Factory Functions) et les modules

Beaucoup de gens s'opposent aux constructeurs car il peut être facile d'introduire des bogues dans le code lors de l'utilisation de constructeurs. L'un des plus gros problèmes avec les constructeurs est qu'ils ressemblent à des fonctions normales mais ne se comportent pas tout comme telles. Il ne faut par exemple par oublier le mot clé *new* sinon le programme ne fonctionnera pas comme prévu.

Il faut surtout retenir que même si les constructeurs ne sont pas nécessairement mauvais, ils ne sont pas le seul moyen, ni le meilleur pour construire des objets.

1. Syntaxe des fonctions d'usine (Factory Functions)

Comparaison entre constructeur et fonction d'usine :

Constructeur :

```
const Person = function(name, age) {
    this.sayHello = () => console.log(`Hello ${name}`);
    this.name = name;
    this.age = age;
};

const jeff = new Person('Jeff', 27);
jeff.sayHello(); // Hello Jeff
```

Fonction d'usine :

```
const personFactory = (name, age) => {
    const sayHello = () => console.log(`Hello ${name}`);
    return { name, age, sayHello };
};

const jeff = personFactory('Jeff', 27);
jeff.sayHello(); // Hello Jeff
```

Pour la fonction d'usine on peut aussi utiliser cela pour la déclaration :

```
const personFactory = function(name, age) {
ou encore :
function personFactory(name, age) {
```

La ligne 3 de la fonction d'usine est la version shorthand de :

```
return {name: name, age: age, sayHello: sayHello};
```

Exemple :

```
const name = "Maynard";
const color = "red";
const number = 34;
const food = "rice";
```

```
console.log(name, color, number, food);
// Maynard red 34 rice
console.log({name, color, number, food});
 // { name: 'Maynard', color: 'red', number: 34, food:
'rice' }
```

2. Les fonctions privées

```
const FactoryFunction = string => {
  const capitalizeString = () => string.toUpperCase();
  const printString = () => console.log(`----${capitalizeString()}----`);
  return { printString };
};
const taco = FactoryFunction('taco');
printString(); // ERROR!!
capitalizeString(); // ERROR!!
taco.capitalizeString(); // ERROR!!
taco.printString(); // this prints "----TACO----"
```

En raison du concept de portée, les fonctions ne sont pas accessibles en dehors de la fonction elle-même. Pour accéder aux fonctions on doit les placer dans le return, et les appeler avec le nom de l'objet créé. C'est pourquoi *taco.capitalizeString()* ne fonctionne pas mais que *taco.printString()* fonctionne. La fonction *printString() étant définie dans FactoryFunction*, elle a donc accès à tout ce qui se trouve à l'intérieur. C'est pourquoi elle fonctionne. C'est le concept de fermeture.

Dans le contexte des fonctions d'usine, le concept de fermeture permet de créer des variables et des fonctions privées qui ne sont utilisées que dans le fonctionnement de objets et pas destinées à être utilisées ailleurs dans le programme. On n'exporte uniquement les fonctions que le reste du programme va utiliser. Dans l'exemple précédent, *printString()* est une fonction publique et *capitalizeString()* est une fonction privée.

Un autre exemple :

```
const Player = (name, level) => {
  let health = level * 2;
  const getLevel = () => level;
  const getName  = () => name;
  const die = () => {
    // uh oh
  };
  const damage = x => {
    health -= x;
    if (health <= 0) {
      die();
    }
  };
  const attack = enemy => {
    if (level < enemy.getLevel()) {
      damage(1);
      console.log(`${enemy.getName()} has damaged ${name}`);
    }
    if (level >= enemy.getLevel()) {
      enemy.damage(1);
      console.log(`${name} has damaged ${enemy.getName()}`);
    }
  };
  return {attack, damage, getLevel, getName};
};
```

```
const jimmie = Player('jim', 10);
const badGuy = Player('jeff', 5);
jimmie.attack(badGuy);
//jim has damaged jeff
```

3. Héritage avec les fonctions d'usine

```
const Person = (name) => {
  const sayName = () => console.log(`my name is ${name}`);
  return {sayName};
}

const Nerd = (name) => {
  // simply create a person and pull out the sayName
function with destructuring assignment syntax!
  const {sayName} = Person(name);
  const doSomethingNerdy = () => console.log('nerd stuff');
  return {sayName, doSomethingNerdy};
}

const jeff = Nerd('jeff');

jeff.sayName(); //my name is jeff
jeff.doSomethingNerdy(); // nerd stuff
```

Ce modèle permet de choisir les fonctions que l'on souhaite inclure dans le nouvel objet.

Si on souhaite inclure toutes les fonctions d'un objet :

```
const Nerd = (name) => {
  const prototype = Person(name);
  const doSomethingNerdy = () => console.log('nerd stuff');
  return Object.assign({}, prototype, {doSomethingNerdy});
}
```

La structure est la suivante : *Object.assign(target, …sources)*

Un effet secondaire utilise de l'encapsulation du fonctionnement internes de nos programmes est l'espacement des noms. C'est une technique utilisée pour éviter les collisions de noms dans nos programmes. Il est donc tout à fait possible de créer une fonction qui ajoute deux nombres, une qui ajoute des éléments à l'affichage HTML, une fonction qui ajoute des éléments à une pile et de les appeler toutes les 3 add si elles sont toutes encapsulées à l'intérieur d'objet : *calculator.add(), displayController.add(), operatorStack.add().*

4. Les modules

Les modules sont très similaires aux fonctions d'usine. La différence réside dans la manière dont ils sont créés :

```
const calculator = (() => {
  const add = (a, b) => a + b;
  const sub = (a, b) => a - b;
  const mul = (a, b) => a * b;
  const div = (a, b) => a / b;
  return {
    add,
    sub,
    mul,
    div,
  };
})();
calculator.add(3,5); // 8
calculator.sub(6,2); // 4
calculator.mul(14,5534); // 77476
```

Le concept est le même que la fonction d'usine. Sauf qu'on ne crée pas des objets que l'on va utiliser ensuite, le module englobe le tout dans une IIFE : Expression de fonction immédiatement invoquée : on écrit une fonction, on la place entre parenthèse, puis on appelle immédiatement la fonction en ajoutant () à la fin.

Puisque l'on n'a pas besoin de faire beaucoup de calculatrice, en la transformant en IIFE on affecte l'objet à la variable *calculator* pour l'utiliser directement.

VI. Gérer la logique d'un programme en JavaScript

https://developer.mozilla.org/fr/docs/Learn/JavaScript/Building_blocks/conditionals

1. Les comparaisons

https://fr.javascript.info/comparison

- < inférieur à ;
- <= inférieur ou égal à ;
- == ou === égal à ;
- >= supérieur ou égal à ;
- > supérieur à ;
- !== différent de

Différence entre == et === :

Il y a deux façons de vérifier si deux valeurs sont égales en JavaScript : == et ===, aussi appelées *égalité **simple*** et *égalité **stricte*** :

- L'égalité simple vérifie la **valeur**, mais pas le type. Donc ceci renvoie la valeur true :
  5 == "5"
- Par contre, l'égalité stricte vérifie à la fois la **valeur et le type**. Donc :
  5 === "5"
  renvoie false , car on compare un number à une string .

De même, il y a deux opérateurs d'inégalité, != et !== , avec la même distinction.

Les opérateurs de comparaison peuvent participer à l'affectation de variable. C'est l'opérateur conditionnel : condition ? exprSiVrai : exprSiFaux

```
var age = n;
var voteable = (age < 18) ? "Too young" : "Old enough";
alert(voteable);
```

En multiple :

```
let age = prompt('age?', 18);

let message = (age < 3) ? 'Hi, baby!' :
  (age < 18) ? 'Hello!' :
  (age < 100) ? 'Greetings!' :
  'What an unusual age!';

alert( message );
```

2. Les instruction if/else :

https://www.w3schools.com/js/js_if_else.asp
https://javascript.info/ifelse

```
let UserLoggedIn = true;
if (UserLoggedIn) {
   console.log("Utilisateur connecté!");
} else {
   console.log("Alerte, intrus!");
}
```

```
if (false) {
  console.log('The code in this block will
not run.');
} else {
  console.log('But the code in this block
will!');
}
// Prints: But the code in this block will!
```

```
if (numberOfGuests == numberOfSeats) {
  // tous les sièges sont occupés
} else if (numberOfGuests < numberOfSeats)
{
  // autoriser plus d'invités
} else {
  // ne pas autoriser de nouveaux invités
}
```

```
const numberOfSeats = 30;
const numberOfGuests = 25;
if (numberOfGuests < numberOfSeats) {
// autoriser plus d'invités
} else {
// ne pas autoriser de nouveaux invités
}
```

Sont évaluées comme *false* les valeurs suivantes : *'', "", 0, null, undefined, NaN (Not a Number).*
N'importe quel chiffre non nul, ou chaîne de caractère non vide sera considéré comme true.

```
let isNightTime = true;

if (isNightTime) {
  console.log('Turn on the lights!');
} else {
  console.log('Turn off the lights!');
}
// Prints Turn on the lights!
```

```
isNightTime ? console.log('Turn on the lights!')
 : console.log('Turn off the lights!');
```

ShortHand Condition :

```
//Short circuiting
if (a>10) {
    doSomething(a);
}
a > 10 && doSomething(a);
```

Initialiser une valeur :

```
//Default Value
if (bar) {
    foo = bar;
} else {
    foo = default;
}

foo = bar != null ? bar : default;
foo = bar || default;
```

Simplifier conditions :

```
if (a === undefined || a ===30 || a ===15 || a === null) {
    // code...
}
if ([undefined, 30, 15, null].includes(a)) {
    // code...
}
if ([undefined, 30, 15, null].indexOf(a) !== -1) {
    // code...
}
if (~[undefined, 30, 15, null].indexOf(a)) {
    // code...
}
```

```
if (a==0) {
    return true;
} else {
    retur  false;
}

return (a===0) ? true : false;
```

La valeurs considérées false (*falsy*) sont "", 0, -0, null, undefinied,NaN. Tout le reste est considéré true (*truthy*).

3. Les opérateurs logiques pour conditions multiples :

https://javascript.info/logical-operators
https://www.digitalocean.com/community/tutorials/how-to-use-the-switch-statement-in-javascript

Pour les conditions multiples, on utilise les opérateurs logiques :

- || – OU logique – pour vérifier si **au moins une** condition est vraie ;

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'The office is closed.' ); // it is the weekend
}
```

OU recherche la 1ère valeur vraie, sinon la dernière

```
let firstName = "";
let lastName = "";
let nickName = "SuperCoder";

alert( firstName || lastName || nickName || "Anonymous"); // SuperCoder
```

On peut alors assigner plus rapidement une variable :

```
let username = '';
let defaultName;
if (username) {
  defaultName = username;
} else {
  defaultName = 'Stranger';
}
console.log(defaultName);
// Prints: Stranger
```

```
let username = '';
let defaultName = username || 'Stranger';

console.log(defaultName); // Prints: Stranger
```

- && – ET logique – pour vérifier si deux conditions sont **toutes les deux** vraies ;

Ou recherche de la 1ère valeur fausse, sinon renvoie la dernière.
En terme priorité && est supérieur à ||

```
a && b || c && d est essentiellement le même que  (a && b) || (c && d).
```

```
let mood = 'sleepy';
let tirednessLevel = 6;
if (mood === 'sleepy' && tirednessLevel > 8) {
  console.log('time to sleep');
} else {
  console.log('not bed time yet');
}
// Prints : not bed time yet
```

- ! – NON logique – pour vérifier si une condition n'est **pas** vraie.

4. L'instruction Switch

```
let groceryItem = 'papaya';
switch (groceryItem) {
  case 'tomato':
    console.log('Tomatoes are $0.49');
    break;
  case 'lime':
    console.log('Limes are $1.49');
    break;
  case 'papaya':
    console.log('Papayas are $1.29');
    break;
  default:
    console.log('Invalid item');
    break;
}
// Prints 'Papayas are $1.29'
```

```
const grade = 87;
switch (true) {
  // If score is 90 or greater
  case grade >= 90:
    console.log("A");
    break;
  // If score is 80 or greater
  case grade >= 80:
    console.log("B");
    break;
  // If score is 70 or greater
  case grade >= 70:
    console.log("C");
    break;
  // If score is 60 or greater
  case grade >= 60:
    console.log("D");
    break;
  default: // Anything 59 or below fail
    console.log("F");
}
// 'B'
```

Attention à ne pas oublier l'instruction break à la fin de chaque *case* sinon JavaScript continuera l'exécution des cas suivants en cascade.

```
// Get number corresponding to the current month, with 0 being January and 11 being December
const month = new Date().getMonth();

switch (month) {
  // January, February, March
  case 0:
  case 1:
  case 2:
    console.log("Winter");
    break;
  // April, May, June
  case 3:
  case 4:
  case 5:
    console.log("Spring");
    break;
  // July, August, September
  case 6:
  case 7:
  case 8:
    console.log("Summer");
    break;
  // October, November, December
  case 9:
  case 10:
  case 11:
    console.log("Autumn");
    break;
  default:
    console.log("Something went wrong.");
}
```

Le switch peut servir dans différents cas mais il est surtout plus facile à lire que l'enchaînement de condition if/else :

```
      if (firstUser.accountLevel === 'normal' ) {
    console.log('You have a normal account!');
} else if (firstUser.accountLevel === 'premium' ) {
    console.log('You have a premium account!');
} else if (firstUser.accountLevel === 'mega-premium' ) {
    console.log('You have a mega premium account!');
} else {
    console.log('Unknown account type!');
}
```

VII. Les fonctions

On déclare une fonction et on précise les arguments nécessaires à la fonction. Ces arguments doivent être précisés lorsqu'on appelle la fonction.

1. Sans arguments

```
greetWorld(); // Output: Hello, World!
function greetWorld() {
  console.log('Hello, World!');
}
```

Une fonction peut être appelée avant sa définition.

2. Avec Arguments

```
// On définit la fonction
function afficherDeuxValeurs(valeur1, valeur2) {
    console.log('Première valeur:' + valeur1);
    console.log('Deuxième valeur:' + valeur2);
}
// On exécute la fonction
afficherDeuxValeurs(12, 'Bonjour');
// > Première valeur:12
// > Deuxième valeur:Bonjour
```

```
function calculateAverageRating(tableau){
    if (tableau.length==0){
      return 0;
    }else{
      let somme = 0;
      for (value of tableau){
        somme += value;
      }
      return (somme/tableau.length);
    }
}
```

3. Avec valeurs par défaut :

```
function makeShoppingList(item1 = 'milk', item2 = 'bread', item3 = 'eggs'){
    console.log(`Remember to buy ${item1}`);
    console.log(`Remember to buy ${item2}`);
    console.log(`Remember to buy ${item3}`);
}
makeShoppingList()
//Output:
//Remember to buy milk
//Remember to buy bread
//Remember to buy eggs
```

Il peut être judicieux de définir les paramètres par défaut lors de l'exécution de la fonction et non pas dans sa déclaration :

```
function showMessage(text) {
    if (text === undefined) {
        // si le paramètre est manquant
        text = 'empty message';
    }
    alert(text);
}
showMessage(); // empty message
```

```
…Ou nous pourrions utiliser l'opérateur || :

function showMessage(text) {
    // if text is undefined or otherwise falsy, set it to 'empty'
    text = text || 'empty';
    alert(text);
}
```

4. Mot clé return

```
function monitorCount(rows, columns){
    return rows * columns
}
const numOfMonitors = monitorCount(4, 5);
console.log(numOfMonitors)  //Print : 20
```

5. Appels de fonction à l'intérieur d'une fonction :

```
function monitorCount(rows, columns) {
    return rows * columns;
}
function costOfMonitors(rows, columns){
    return monitorCount(rows, columns) * 200
}
const totalCost = costOfMonitors(5, 4)
console.log(totalCost) //print 4000
```

6. Expression de fonction :

```
const plantNeedsWater = function(day){
    if (day === 'Wednesday') {
      return true;
    } else {
      return false;
    }
};
console.log(plantNeedsWater('Tuesday')) //Print false
```

7. Fonction fléchée :

```
const rectangleArea = (width, height) => {
    return width * height;
};
let area = rectangleArea(4, 8)
console.log(area); //Print 32
```

8. Simplification :

```
// ZERO PARAMETERS
const functionName = () => {};
// ONE PARAMETERS
const functionName = paramOne => {};
// TWO OR MORE PARAMETERS
const functionName = (paramOne, paramTwo) => {};
// SINGLE-LINE BLOCK
const sumNumbers = number => number + number;
ex : let sum = (a, b) => a + b;
// MULTI-LINE BLOCK
const sumNumbers = number => {
  const sum = number + number;
  return sum;
};
```

9. Fonctions en paramètre d'une autre fonction :

```
const addTwo = num => {
    return num + 2;
}
const checkConsistentOutput = (func, val) => {
    let checkA = val + 2;
    let checkB = func(val);
    if (checkA === checkB) {
      return checkA;
    } else {
      return 'inconsistent results';
    }}
```

10. arguments

```
function func1(a, b, c) {
    console.log(arguments[0]); // first argument
    // expected output: 1
    console.log(arguments[1]); // second argument
    // expected output: 2
    console.log(arguments[2]); // third argument
    // expected output: 3
  }
func1(1, 2, 3);
```

Cela peut être intéressant quand les fonctions sont appelées avec plus d'arguments que la définition ou pour passer un nombre variable d'argument :

```
function longestString() {
    let longest = '';
    for (let i = 0; i < arguments.length; i++) {
      if (arguments[i].length > longest.length) {
        longest = arguments[i];
      }
    }
    return longest;
}
```

11. *...Rest* paramètre

Cela autorise une fonction à accepter un nombre infini d'arguments

```
function sum(...theArgs) {
    let total = 0;
    for (const arg of theArgs) {
      total += arg;
    }
    return total;
}
console.log(sum(1, 2, 3)); // expected output: 6
console.log(sum(1, 2, 3, 4));// expected output: 10
```

Il ne peut y avoir qu'un seul *...restParam*. S'il y a des arguments en plus le *...restParam* doit être en dernier.

Attention au scope des variables ! En JavaScript, les variables ne peuvent être vues qu'à l'intérieur du bloc de code dans lequel elles sont déclarées.

```
const city = 'New York City';

function logCitySkyline() {
  let skyscraper = 'Empire State Building';
  return 'The stars over the ' + skyscraper + ' in ' + city;
}
console.log(logCitySkyline());
console.log(skyscraper); // ReferenceError
```

Comme la variable *city* a été déclarée à l'extérieur d'un block, elle est accessible par n'importe quelle partie du code comme la fonction *logCitySkyline*. On appelle ça une variable globale.

Mais on ne peut pas accéder à la variable *skyscraper* en dehors de la fonction. Elle est définie à l'intérieur d'un block, elle n'est donc accessible que dans ce block.

Il faut éviter d'abuser des variables globales pour ne pas polluer l'espace de nommage et avoir des conflits avec des variables locales.

VIII. Les boucles

1. Les boucles FOR

```
// Pour afficher les nombres de 5 à 10 :
for (let number = 5; number <= 10; number++) {
    console.log(number);
 }
```

```
// The loop below loops from 0 to 3. Edit it to loop backwards from 3 to 0
for (let counter = 3; counter >= 0; counter--) {
    console.log(counter);
  }
```

```
// Boucle à l'intérieur d'un tableau à l'aide des indices
const vacationSpots = ['Bali', 'Paris', 'Tulum'];
for (let i = 0; i < vacationSpots.length; i++) {
  console.log('I would love to visit ' + vacationSpots[i]);
}
```

- Boucles imbriquées

```
// Pour vérifier s'il y a des éléments communs dans deux tableaux :
const bobsFollowers = ['Pierre', 'Rémi', 'Louis', 'Benjamin'];
const tinasFollowers = ['Jeanne', 'Pierre', 'Benjamin'];
const mutualFollowers  = [];
for (let i = 0; i < bobsFollowers.length; i++){
  for (let j = 0; j < tinasFollowers.length; j++){
    if (bobsFollowers[i] === tinasFollowers[j]){
      mutualFollowers.push(bobsFollowers[i]);
    }
  }
}
console.log(mutualFollowers) //Output : [ 'Pierre', 'Benjamin' ]
```

- Boucles *for/in*

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in

- Tableau

```
const passengers = [
    "Will Alexander",
    "Sarah Kate'",
    "Audrey Simon",
    "Tao Perkington"
]
```

```
for (let i in passengers){
    console.log("Embarquement du passager " + passengers[i]);
}
```

La boucle *for/in* itère sur les index du tableau. On accède donc au valeur grâce aux crochets.

- Objet

```
const person =
{fname:"John",
    lname:"Doe",
    age:25};
```

```
for (let x in person) {
    console.log(x); //fname  lname  age
  }
```

La boucle *for/in* itère sur les clés d'un objet.

- Boucles *for/of*

- Tableau

```
const passengers = [
    "Will Alexander",
    "Sarah Kate'",
    "Audrey Simon",
    "Tao Perkington"
]
```

```
for (let passenger of passengers){
    console.log("Embarquement du passager " + passenger);
}
```

La boucle *for/of* itère directement sur les valeurs d'un tableau.

- String

```
let language =
"JavaScript";
```

```
for (let x of language) {
    console.log(x); //J a v a ...
}
```

2. Les boucles WHILE

- *While*

```
// A while loop that prints 1, 2, and 3
let counterTwo = 1;
while (counterTwo < 4) {
  console.log(counterTwo);
  counterTwo++;
};
```

```
const cards = ['diamond', 'spade', 'heart', 'club'];
// Code qui boucle tant qu'on ne tombe pas sur 'spade'
let currentCard;
while (currentCard !== 'spade') {
  currentCard = cards[Math.floor(Math.random() * 4)];
  console.log(currentCard);
};
```

```
let seatsLeft = 10;
let passengersStillToBoard = 8;
let passengersBoarded = 0;

while (seatsLeft > 0 && passengersStillToBoard > 0) {
    passengersBoarded++; // un passager embarque
    passengersStillToBoard--; // donc il y a un passager de moins à embarquer
    seatsLeft--; // et un siège de moins
}
console.log(passengersBoarded); // imprime 8, car il y a 8 passagers pour 10 sièges
```

- *Do….While*

```
let countString = '';
let i = 0;
do {
  countString = countString + i;
  i++;
} while (i < 5);
console.log(countString); // Output 01234
```

```
const firstMessage = 'I will print!';
const secondMessage = 'I will not print!';
// A do while with a stopping condition that evaluates to false
do {
 console.log(firstMessage)
} while (true === false);
// A while loop with a stopping condition that evaluates to false
while (true === false){
  console.log(secondMessage)
};
// Output I will print!
```

La différence avec une boucle WHILE c'est la séquence est réalisée au moins une fois, même si la condition est fausse

3. Le mot clé *break*

```
for (let i = 0; i < 99; i++) {
    if (i > 2 ) {
       break;
    }
    console.log('Banana.');
}
console.log('Orange you glad I broke out the loop!');
```

Il permet d'arrêter la boucle : ici après la 3e itération :

IX. Les itérateurs

- .forEach() prend en argument une fonction.

```
const artists = ['Picasso', 'Kahlo', 'Matisse', 'Utamaro'];
// Réalise le block de code pour tous les termes de la liste
artists.forEach(artist => {
  console.log(artist + ' is one of my favorite artists.');
});
```

- .map() prend en argument une fonction.

```
const numbers = [1, 2, 3, 4, 5];
// Retourne une liste transformée par le block de code
const squareNumbers = numbers.map(number => {
  return number * number;
});
console.log(squareNumbers); // Output [ 1, 4, 9, 16, 25 ]
```

- .filter() prend en argument une fonction.

```
const things = ['desk', 'chair', 5, 'backpack', 3.14, 100];
// Retourne une liste correspond à la condition
const onlyNumbers = things.filter(thing => {
  return typeof thing === 'number';
});
console.log(onlyNumbers); //Output [ 5, 3.14, 100 ]
```

- .findIndex() prend en argument une fonction.

```
const jumbledNums = [123, 25, 78, 5, 9];
// Retourne l'index du 1er élément qui vérifie la condition
const lessThanTen = jumbledNums.findIndex(num => {
  return num < 10;
});
console.log(lessThanTen) //utput 3
```

- .find()

Même principe que .findIndex() mais retourne la première valeur qui vérifie la condition.

- .reduce()

```
const newNumbers = [1, 3, 5, 7];
// Retour une seule valeur après avoir parcouru le tableau
const newSum = newNumbers.reduce((accumulator, currentValue) =>
{
  console.log('The value of accumulator: ', accumulator);
  console.log('The value of currentValue: ', currentValue);
  return accumulator + currentValue
},10)
console.log(newSum); //OutPut 26
```

```
The value of
accumulator:  10
The value of
currentValue:  1
The value of
accumulator:  11
The value of
currentValue:  3
The value of
accumulator:  14
The value of
currentValue:  5
The value of
accumulator:  19
The value of
currentValue:  7
```

Trouver le maximum ou le minimum d'un tableau :

```
const numbers = [1, 10, 4, 5, 7];
const maxVal = numbers.reduce((a, b) => {
    return a > b ? a : b
});
console.log(maxVal); //10
const minVal = numbers.reduce((a, b) => {
    return a < b ? a : b
});
console.log(maxVal); //1
```

- .reduceRight()

Même chose que .reduce() mais itère le tableau de la droite vers le gauche sans avoir besoin de faire un .reverse().

- .some()

```
// Retourne true or false si au moins 1 éléments respecte la condtion
const words = ['unique', 'uncanny', 'pique', 'oxymoron', 'guise'];
console.log(interestingWords.every(word => word.length > 5 ));
```

- .every()

Même chose que .some() mais avec tous les éléments du tableau.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/every

X. Gérer les erreurs et les exceptions :

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors

1. Les erreurs de syntaxes :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/SyntaxError

Une faute d'écriture : un oubli de crochet, d'accolade, de point-virgule, une faute d'orthographe sur un mot clé, une variable, une fonction. Facile à corriger, mais parfois difficile à trouver.

2. Les erreurs logiques :

Ces erreurs ont lieu si on affecte une valeur erronée à une variable, une erreur dans les conditions dans des instructions *if*, un ordre incorrect d'écriture des lignes de code. Ces erreurs sont plus difficiles à trouver et à corriger parce que le code n'est pas faux. Il ne fait simplement pas ce que vous souhaitez qu'il fasse.

```
//On simule avoir 22 ans donc être majeur
const monAge = 22;

if(monAge < 18) {
    console.log("vous êtes majeur");
}else{
    console.log("vous êtes mineur");
}
```

3. Les erreurs d'exécution :

Elles surviennent quand quelque chose d'inattendu se produit dans notre application. Il s'agit souvent de quelque chose associé aux ressources extérieurs (connexions réseau, appareils physiques, etc…) ou à une saisie/erreur humaine.

Dans ces situations, on peut prévoir du code de traitement d'erreur. De cette façon l'erreur ne fera pas planter notre programme et pourra être corrigée.

```
if (dataExists && dataIsValid) {
  // utiliser les données ici
} else {
  // gérer l'erreur ici
}
```

```
try {
  // code susceptible à l'erreur ici
} catch (error) {
  // réaction aux erreurs ici
}
```

4. ReferenceError

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/ReferenceError

Arrive lorsque l'on fait référence à une variable qui n'est pas déclarée/initialisée dans sa portée actuelle.

5. TypeError

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/TypeError

Arrive quand un argument passé à une fonction n'est pas dans le type attendu par cette fonction. Idem avec opérateur.

Tentative de modification d'une valeur qui ne peut pas être changée

Utilisation d'une valeur d'une manière inappropriée

XI. Ecrivez du code propre et facile à maintenir

Le but des fonctions est d'être plus efficace et de ne pas se répéter. DRY Don't Repeat Yourself

```
if (firstUser.online) {
if (firstUser.accountType === "normal") {
      console.log("Hello " + firstUser.name
+ "!");
} else {
    console.log("Welcome back premium user
" + firstUser.name + "!");
}

}
if (secondUser.online) {
if (secondUser.accountType === "normal") {
    console.log("Hello " + secondUser.name
+ "!");
} else {
    console.log("Welcome back premium user
" + secondUser.name + "!");
}
}

if (thirdUser.online) {
if (thirdUser.accountType === "normal") {
    console.log("Hello " + thirdUser.name +
"!");
} else {
    console.log("Welcome back premium user
" + thirdUser.name + "!");
}
}
```

```
const sendWelcomeMessageToUser = (user) =>
{
if (user.online) {
if (user.accountType === "normal") {
        console.log("Hello " + user.name +
"!");
} else {
        console.log("Welcome back premium
user " + user.name + "!");
}
}
}

sendWelcomeMessageToUser(firstUser);

sendWelcomeMessageToUser(secondUser);

sendWelcomeMessageToUser(thirdUser);
```

1. Nommer les fonctions par ce qu'elles font.
2. Décomposer les fonctions au maximum
3. Nommer les variables par ce qu'elles représentent, en cameCase
4. Ne pas oublier les points virgules
5. Respecter l'indentation
6. Ligne de code de 80 caractères maximum

"La première règle des fonctions est qu'elles devraient être petites. La deuxième règle des fonctions est qu'elles devraient être encore plus petites"

Robet C. Martin, Clean Code : A Handbook of Agile Software Craftsmanship

Il est aussi important de laisser des commentaires pour expliquer ce que l'on fait :

La deuxième version est plus longue mais est plus claire. Car chaque fonction ne fait qu'une seule chose, mentionnée dans leur nom.

Pour avoir un code clair, il peut être important de bien réfléchir la logique en pseudo-code

Version 1 :

```
const printStringStats = (stringToTest) => {
    const wordArray = stringToTest.split(" ");
    const wordCount = wordArray.length;
    let letterCount = 0;

    for (let word of wordArray) {
        word.replace(/[.,\/#!$%\^&\*;:{}=\-_`~()]/g, "");
        letterCount += word.length;
    }

    const averageWordLength = parseFloat((letterCount / wordCount).toFixed(2));
    const stringStats = {
        wordCount: wordCount,
        letterCount: letterCount,
        averageWordLength: averageWordLength
    };

    console.log(stringStats);
}
```

Version 2 :

```
const getWordCount = (stringToTest) => {
    const wordArray = stringToTest.split(' ');
    return wordArray.length;
}

const getLetterCount = (stringToTest) => {
    const wordArray = stringToTest.split(' ');
    let totalLetters = 0;
    for (let word of wordArray) {
        // retire la ponctuation pour ne compter que les lettres
        word.replace(/[.,\/#!$%\^&\*;:{}=\-_`~()]/g, "");
        totalLetters += word.length;
    }
    return totalLetters;
}
/*
** renvoie la longueur moyenne des mots
** arrondie à deux chiffres après la virgule
*/
const getAverageWordLength = (stringToTest) => {
    return parseFloat((getLetterCount(stringToTest) /
getWordCount(stringToTest)).toFixed(2));
}

const printStringStats = (stringToTest) => {
    console.log({
        wordCount: getWordCount(stringToTest),
        letterCount: getLetterCount(stringToTest),
        averageWordLength: getAverageWordLength(stringToTest)
    })
}
```

Code non DRY :

```
//variable de différentes personnes
let personne1 = "Jean";
let personne2 = "Paul";
let person3 = "Marcel";

//On met la première lettre en majuscule, on salue la première personne et on donne le nombre de lettre dans son prénom
personne1 = personne1[0].toUpperCase() + personne1.substr(1);
const longueurPrenom1 = personne1.length;
console.log(`Bonjour ${personne1}, ton prénom contient ${longueurPrenom1} lettres`);

//On met la première lettre en majuscule, on salue la deuxième personne et on donne le nombre de lettre dans son prénom
personne2 = personne2[1].toUpperCase() + personne2.substr(1);
const longueurPrenom2 = personne2.length;
console.log(`Bonjour ${personne2}, ton prénom contient ${longueurPrenom2} lettres`);

//On met la première lettre en majuscule, on salue la troisième personne et on donne le nombre de lettre dans son prénom
personne3 = personne3[2].toUpperCase() + personne3.substr(1);
const longueurPrenom3 = personne3.length;
console.log(`Bonjour ${personne3}, ton prénom contient ${longueurPrenom3} lettres`);
```

Code mal nommé :

```
//tableau des âges des élèves dans la classe

const lrf = [14, 14, 15, 14, 16, 14, 14, 13];

// Nombre d'élèves

const kf = lrf.length;

// variable pour calculer la somme des âges

let mf = 0;

for(let df of lrf){

    mf += df;

}

//moyenne d'âge dans la classe

const mld = mf / kf;

console.log('Il y a ' + kf + " élèves dans la classe et la moyenne d'âge est " + mld);
```

Code après correction :

```
//variable de différentes personnes
let personne1 = "Jean";
let personne2 = "Paul";
let personne3 = "Marcel";

function saluer(prenom){
  //On met la première lettre en majuscule, on salue la personne et on donne le nombre de lettre dans son prénom
  const monPrenom = prenom[0].toUpperCase() + prenom.substr(1);
  const longueurPrenom = monPrenom.length;
  console.log(`Bonjour ${monPrenom}, ton prénom contient ${longueurPrenom} lettres`);
}

//On salue les 3 personnes
saluer(personne1);
saluer(personne2);
saluer(personne3);
```

ou on peut rajouter

```
let personnes = [personne1, personne2, personne3];

for (let personne of personnes){
  saluer(personne)
}
```

Code après correction :

```
//tableau des âges des élèves dans la classe

const agesElevesDeClasse = [14, 14, 15, 14, 16, 14, 14, 13];

// Nombre d'élèves

const nombreEleves = agesElevesDeClasse.length;

// variable pour calculer la somme des âges

let sommeAges = 0;

for(let age of agesElevesDeClasse){

    sommeAges += age;

}

//moyenne d'âge dans la classe

const moyenneAgesDeClasse = sommeAges / nombreEleves;

console.log('Il y a ' + nombreEleves + " élèves dans la classe et la moyenne d'âge est " + moyenneAgesDeClasse);
```

Code mal mis en forme :

```
const temperature = 25;



if(temperature < 10){  console.log("Il fait
très froid"); }

else if(temperature < 0){

console.log(

"Il fait froid"

);

}else if(temperature < 10){

console.log(            "Il fait frais");

}else if(temperature < 20){

console.log("Il fait doux");

}else if(temperature < 30){

console.log("Il fait bon");









}else{









console.log("Il fait chaud");

}
```

```
const temperature = 25;



if(temperature < 10){

    console.log("Il fait très froid"); }

else if(temperature < 0){

    console.log("Il fait froid");

}else if(temperature < 10){

    console.log("Il fait frais");

}else if(temperature < 20){

    console.log("Il fait doux");

}else if(temperature < 30){

    console.log("Il fait bon");

}else{

    console.log("Il fait chaud");

}
```

Résumé :

- Il faut factoriser le code en fonction pour ne pas se répéter, ou quand une fonction fait trop de chose.
- Les fonctions qui ne font qu'une seule chose sont plus claires que celles qui en font plusieurs.
- Laisser des commentaires permet une meilleure compréhension du code.
- Il est important de nommer les noms et les fonctions avec des noms claires et descriptifs.

Tester qu'une fonction fait ce qu'elle dit :

Les tests permettent de déceler les erreurs.

- Tests unitaires

Il vérifie des unités individuelles (fonctions uniques ou classes) en leur fournissant une entrée et en s'assurant qu'elles donnent la sortie attendue. Une fonction qui n'a qu'un seul usage est donc plus facile à tester.

```
const getWordCount = (stringToTest) => {
    const wordArray = stringToTest.split('
');
    return wordArray.length;
}

const getLetterCount = (stringToTest) => {
    const wordArray = stringToTest.split('
');
    let totalLetters = 0;
    for (let word of wordArray) {
        word.replace(/[.,\/#!$%\^&\*;:{}=\-
_`~()]/g, "");
        totalLetters += word.length;

    }

    return totalLetters;
}
```

```
const testSimpleWordCount = () => {
    const testString = 'I have four
words!';
    if (getWordCount(testString) !== 4) {
        console.error('Simple getWordCount
failed!');
    }
}

const testEdgeWordCount = () => {
    const testString = '             ';
    if (getWordCount(testString) !== 0) {
        console.error('Edge getWordCount
failed!');
    }
}

const testSimpleLetterCount = () => {
   const testString = 'I have twenty one
letters!';
    if (getLetterCount(testString) !== 21)
{
        console.error('Simple
getLetterCount failed!');
    }
}

const testEdgeLetterCount = () => {
    const testString = '")(&;//!!';
    if (getLetterCount(testString) !== 0) {
        console.error('Edge getLetterCount
failed!');
    }
}
```

Des architectures de test permettent d'automatiser les tests à l'aide de fonctions et de syntaxe spécifiques. Ex :

```
describe('getWordCount()', function() {
    it('should find four words', function() {
        expect(getWordCount('I have four words!').to.equal(4));
    });

    it('should find no words', function() {
        expect(getWordCount('      ').to.equal(0));
    });
});
```

- Les tests d'intégration

Ils permettent de s'assurer que les multiples fonctions travaillent ensemble comme elles sont censées le faire.

- Les tests fonctionnels

Ils vérifient les scénarios complets en contexte. Ex : un utilisateur se connecte, ouvre ses notifications, les marque toutes comme lues. Ils vérifient aussi les ressources externes que le projet utilise comme par exemple un système de paiement tiers.

Déboguer son code :

- La console

Des affichages dans la console permettent de voir comment agit la fonction. Cela convient pour des cas simples et isolés mais dans les projets plus complexes c'est plus difficile et chronophage. Il faut utiliser des outils plus puissants.

- Les outils pour développeur

Ils sont présents dans les navigateurs (chrome, edge, firefox, safari…). Ils ont un système qui permet d'ajouter des points d'arrêt (breakpoint). Ce qui permet de parcourir l'exécution ligne après ligne en vérifiant les valeurs des variables à chaque étape.

La plupart des environnements de développement intégrés (Visual studio code, Webstorm) comportent aussi un débogueur. Cela peut être pratique si le code ne doit pas s'exécuter dans une page web.

Les fonctions récursives :

On fonction récursive est une fonction qui s'appelle elle-même. Attention aux fonctions récursives qui peuvent continuer à s'appeler à l'infini. Une fonction récursive a besoin d'un cas de base ou *base case* pour savoir quand son travail est terminé.

```
function factorielle(number){
    if (number == 1){
      return 1
    }else{
      return number*factorielle(number-1)
    }
}
```

Les méthodes d'instances et les propriétés :

```
class BankAccount {
  constructor(owner, balance) {
      this.owner = owner;
      this.balance = balance;
  }

  showBalance() {
      console.log("Solde: " + this.balance
+ " EUR");
  }

  deposit(amount) {
      console.log("Dépôt de " + amount + "
EUR");
      this.balance += amount;
      this.showBalance();
  }

  withdraw(amount) {
      if (amount > this.balance) {
                console.log("Retrait refusé
!");
      } else {
          console.log("Retrait de " +
amount + " EUR");
          this.balance -= amount;
          this.showBalance();
      }
  }
}
```

On initialise la classe comme vu précédemment.

Une méthode d'instance est une fonction à l'intérieur d'une classe.

```
const newAccount = new BankAccount("Will
Alexander", 500);

newAccount.showBalance(); // imprime
"Solde: 500 EUR" à la console
```

Les méthodes statiques :

Les méthodes statiques, contrairement aux méthodes d'instances, ne sont pas liées à des instances de classe particulière.

Ici, on n'a pas besoin de créer à l'aide de *new* une instance de l'objet *Math* pour utiliser ces méthodes.

```
const randomNumber = Math.random(); // crée
un nombre aléatoire sur l'intervalle [0, 1]

const roundMeDown = Math.floor(495.966); //
arrondit vers le bas à l'entier le plus
proche, renvoie 495
```

```
class BePolite {
```

Dans cette classe, on n'a pas besoin d'ajouter un *constructor* car on ne va pas l'instancier.

```
  static sayHello() {
      console.log("Hello!");
  }
  static sayHelloTo(name) {
      console.log("Hello " + name + "!");
  }
  static add(firstNumber, secondNumber) {
      return firstNumber + secondNumber;
  }
}

BePolite.sayHello(); // imprime "Hello!""

BePolite.sayHelloTo("Will"); // imprime
"Hello Will!""

const sum = BePolite.add(2, 3); // sum = 5
```

Toutes ces fonctionnalités pourraient être des fonctions mais l'avantage d'utiliser des méthodes de classe statiques est par exemple de pouvoir les regrouper par catégorie ou par type.

XII. Javascript trick

```
//Copy to Clipboard
const copyToClipboard = (text) => navigator.clipboard &&
navigator.clipboard.writeText && navigator.clipboard.writeText(text);

copyToClipboard("Hello World")
```

```
//Detect Dark Theme
const isDarkMode = () => window.matchMedia &&
window.matchMedia("(prefers-colors-scheme: dark)").matches;
console.log(isDarkMode())
```

```
// scroll to top
const scrollToTop = (element) => element.scrollIntoView({
    behavior: "smooth", block: "start"});
```

```
// scroll to Bottom
const scrollToBottom = (element) => element.scrollIntoView({
    behavior: "smooth", block: "end"});
```

```
// DEEP COPY OBJECT
const deepCopy = obj => JSON.parse(JSON.stringify(obj))
```

```
//Générer une couleur au hasard
const generateRandomHexColor = () => {
    return`#${Math.floor(Math.random() * 0xffffff).toString(16)}`;
}
generateRandomHexColor()
```

STOP using console.log() !

Console.dir(element(DOM)) affiche les propriétés de l'objet JS

Console.warn("This is a warning ARGH!!")

Console.info("This is an information message | Connection Valid");

Console.error("Something went wrong!!")

Console.clear() : nottoie la console

Console.table(array ou objet) affiche le tableau ou l'objet comme un tableau.

Console.group("name : string") créer des messages console.log() indenté jusqu'à console.groupeEnd(("name : string")

Console.time("name") puis console.timeEnd("name") crée un chronomètre et affiche le temps à la fin.

Console.count("name") indique le nombre de fois que ce console.count("name") a été appelé.

Console.assert() écrit un message d'erreur dans la console si l'assertion est fausse. Si l'assertion est vrai, rien ne se passe.

```
for (let i=0; i <= 5; i++) {
    console.log('the number is ' + i);
    console.assert(i%2 === 0, {number i, errorMsg: 'The number is not event'});
}
//Affichera pour les nimbres impairs
//Assertion failed: {number: 1, errorMsg; 'The number is not even'}
console.assert(document.getElementById("adress"), "'adress' Not found")
```

```
//Error cause
try {
    //something
} catch (error) {
    throw new error ('something went wrong', {cause: error});
}

// Await can be used outside of async block
if (userIsLoggedIn) {
    await getUsersPreferences();
}
```

```
// Check apple device
const checkAppleDevice = () =>
/Mac|iPod|iPhone|iPad/.test(navigator.platform);
console.log(checkAppleDevice()); // true/false
```

```
// Check if Tab Active
const isTabActive = () => !document.hidden; //not hidden
isTabActive(); // true / false
```

```
// Scroll To The Top
const scrollToTop = () => window.scrollTo(0, 0);
```

I. Retenez les notions du cours

Les classes :

```
class MaClass{ }
```

Instance de classe :

```
let monInstance = new MaClass();
```

Propriété/attribut de classe :

```
class Maison{

    constructor(couleur){
        this.couleur = couleur;
    }

}

let maMaison = new Maison('rouge');
console.log(maMaison.couleur) // Donnera "rouge"
```

Méthode de classe :

```
class Maison{

    constructor(couleur){
        this.couleur = couleur;
    }

    changerCouleur(nouvelleCouleur){
        this.couleur = nouvelleCouleur;
    }

}

let maMaison = new Maison('rouge');
console.log(maMaison.couleur) // Donnera "rouge"
maMaison.changerCouleur('bleu')
console.log(maMaison.couleur) // Donnera "bleu"
```

Les exceptions :

```
try {

    fonctionQuiRetourneUneException();

} catch(e) {

    console.log("il y a une Exception: "+e.getMessage());

}
```

Récursivité :

```
function factorielle(number){

    if(number <= 1) return 1;

    else return (number * factorielle(number-1));

}
```