JavaScript :)

Maj: 03/05/25

```
<!DOCTYPE html>
<html lang="fr">
<head>
<meta name="Author" content="Mda"><meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<title> --Titre-- </title>
<script src="jquery-3.5.1.min.js"></script>
<script type="text/javascript" src="jquery.maphilight.min.js"></script>
<script type="text/javascript" async src="imageMapResizer.min.js" ></script>
<script type="text/javascript" defer src="carrouselv2.js"  ></script>
<link rel="stylesheet" href="carrouselv2.css" media="all">
</head>
```

Bien noter la présence des attributs **async** ou **defer** à l'intérieur des balises scripts.
Leur but est principalement de charger et lancer l'interprétation de code JavaScript **sans bloquer le rendu HTML** (son affichage à l'écran). Ils ne concernent en général que des interprétations de codes situés dans des fichiers externes (lorsque l'attribut `src` est utilisé) et viennent assouplir la pratique communément admise de placer - dans la mesure du possible - les balises `<script>` à la fin du document juste avant la fermeture de `</body>`

- **async** : charger/exécuter les scripts de façon asynchrone.
- **defer** : différer l'exécution à la fin du chargement du document.

InnerHTML

```
document.getElementById("texte_b").innerHTML = "La banque a : <b>"+ncarte_b+"</b>";
<div id="part4"><p id="texte_b"></p></div>
```

```
OU pour insérer le texte dans le paragraphe <p> qui utilise la classe "texte" :

document.querySelector("p.texte").innerHTML = "La banque est pourrie."

OU pour sélectionner le premier paragraphe dont le parent est une <div> :

let select_p = document.querySelector("div > p");
```

console.log

```
console.log(obj1 [, obj2, ..., objN]);
console.log("blabla : " + bla_var);
```

Chaine

```
var s="Bonjour" ;
document.write(s.charAt(2)) ; // retourne n
document.write(s.indexOf("jo")) ; // retoune 3
```

Méthodes

```
replace(expr_reg, nouvelle_chaine)  let text="Visit Mic!"; let result=text.replace("Mic", "W3Schools");
search(expr_reg)
slice(debut[, fin])
split(separateur[, limite])  var chaine="Jean-Paul/Arthur/Léon"; var tableau=chaine.split("/");
substr(debut[, taille])      var chaine="Jean-Paul/Arthur/Léon"; var subChaine = chaine.substr(5,2)
substring(debut, fin)
toLowerCase(), toUpperCase()
```

Opérateur pour concaténer : +

```
document.write("11.3"+5) // retourne 11.35
```
Cast <string>→<float> : `document.write(parseFloat("11.3")+5) // retourne 16.3`

Tableaux

```
var tab1 = new Array(2) ; // déclaration tableau 2 éléments
tab1[35] = 'A' ; // la taille est modifiée automatiquement.
```

```
Var tab2 = new Array ('A', 3, 4) ;
For (i in tab2) { window.alert("tab2["+i+"]  : "+tab2[i]) ; }
```

```
// Adding a single element:
const cart = ['apple', 'orange']; cart.push('pear'); // [apple', 'orange', 'pear']
// Adding multiple elements:
const numbers = [1, 2]; numbers.push(3, 4, 5);
// Pop() : remove last element  (valeur de retour : l'élément supprimé)
numbers.pop(); // [3, 4]
// Shift() : remove element from the beginning
numbers.shift(); // [4]
// Unshift() : add item to the beginning
numbers.unshift(3); // [3, 4]
```

Propriétés : length (taille)
Si `const tab1 = ['a','b','c']` => `tab1=[]` ne marche pas pour vider le tableau car const.
`tab1.length = 0` ==> OK pour vider.
…

Méthodes

concat(tab2, tab3, …)	join(sépar)	pop()	push(val1, val2, …)	shift()
unshift(val1, val2, …)	slice(début[, fin])			

Transtypage

Cast Date > String

```
const d = new Date();
let text = d.toString();
```

Cast Int > String

```
const i = 25;
let text = String(i); // ou string.valueOf(i)
```

Cast String > Float

```
const str = ('3.26');
let f = parseFloat(str); // ou Number(str)
```

Cast String > Int

```
const str = ('3.26');
let i = Number(str); // attention à la majuscule
```

Cast Float > Int

```
const f = 3.26;
let i = parseInt(f); // retourne 3
```

Undeclared variables (created without a keyword **`var`, `let`, `const`**), are always **global**, even if they are created inside a function.

Vérifier le type d'une variable :

```
typeof 32.5 // returns 'number'
Alternative syntax, like a function :
typeof(32.5)
typeof(20 * x)
```

```
console.log("type xhr1 objet: ", xhr1 instanceof Object);
…
getXhr_var = getXhr();
textAlert = (Object.prototype.toString.call(getXhr_var) == '[object Object]') ? "type Objet" : "non Object"
```

Opérateur de coalescence "null" :

opérateur logique qui renvoie son opérande de droite lorsque son opérande de gauche vaut `null` ou `undefined` et qui renvoie son opérande de gauche sinon.

```
const foo = null ?? "default string";
console.log(foo);  // Expected output: "default string"
const baz = 0 ?? 42;
console.log(baz);  // Expected output: 0
```

```
const c=2; let res = c ?? "non défini ou null"; console.log(`res : ${res}`);
// retourne        res : 2
let c; let res = c ?? 0; console.log(`res : ${res}`);
// retourne        res : 0
```

Maths

```
Math.floor(x)          +grand entier < x       ex: 5.93->5       5.05->5
Math.ceil(x)           +petit entier > x       ex: 0.95->1       7.2->8
Math.round(x)          arrondi : entier le + proche       ex: 5.93->6
```

`Math.random()` renvoie un nombre dans l'intervalle [0; 1[

```
Array(4).fill(2)  // [2,2,2,2]
```

`map` `((x,i) => (x+2*i))` transformation de chaque élément d'un tableau, avec x la valeur de l'élément, et i l'index de l'élément dans le tableau.

```
Array(4).fill(1).map((x,i) => (x+3*i)).join('.')     // 1.4.7.10
```

```
const numbers = [4, 9, 16, 25];
const newArr = numbers.map(Math.sqrt)
// [2,3,4,5]
```

document.querySelector

La méthode `querySelector()` de l'interface "document" retourne **le premier élement** correspondant au sélecteur (ou groupe de sélecteurs) spécifié(s), ou `null` si aucune correspondance n'est trouvée.

```
element = document.querySelector(sélecteurs);
```

Dans cet exemple, le premier élément dans le document qui contient la classe "`maclasse`" est retourné :

```
var el = document.querySelector(".maclasse");
```

Un sélecteur plus complexe

Les *sélecteurs* peuvent également être réellement puissants comme le montre l'exemple suivant. Ici, le premier élément `<input name="identifiant"/>` dans un `<div class="panneau-utilisateur principal">` dans le document est retourné :

```
<div class="panneau-utilisateur principal">
<input name="identifiant"/>
```

```
var el = document.querySelector("div.panneau-utilisateur.principal input[name='identifiant']");
```

Document.querySelectorAll()

Pour obtenir une `NodeList` (*liste de noeuds*) de tous les éléments `<p>` dans le document :

```
const matches = document.querySelectorAll("p");
```

Cet exemple renvoie la liste de tous les éléments `div` du `document` dont l'attribut de classe a pour valeur "`note`" ou "`alert`" :

```
const matches = document.querySelectorAll("div.note, div.alert");
```

Ici est créé un nouveau qui est inséré avant l'élément avec l'identifiant "div1".

HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>||Working with elements||</title>
</head>
<body>
  <div id="div1">The text above has been created dynamically.</div>
</body>
</html>
```

JavaScript

```
document.body.onload = addElement;
function addElement () {
  // crée un nouvel élément div
  var newDiv = document.createElement("div");
  // et lui donne un peu de contenu
  var newContent = document.createTextNode('Hi there and greetings!');
  // ajoute le nœud texte au nouveau div créé
  newDiv.appendChild(newContent);
  // ajoute le nouvel élément créé et son contenu dans le DOM
  var currentDiv = document.getElementById('div1');
  document.body.insertBefore(newDiv, currentDiv);
}
```

==========

Math + contrôle de sélection

```
function decimalAdjust(type, value, exp) {
  type = String(type);
  if (!["round", "floor", "ceil"].includes(type)) {
    throw new TypeError(
      "The type of decimal adjustment must be one of 'round', 'floor', or 'ceil'."
    );
  }
….
}
```

Fonction fléchée

```
let somme = (a,b) => a+b;
let reponse = () => {  console.log("test"); console.log("test2") }
// utilisation :
let c = somme(4,6);
```

Expression de fonction

```
let somme = function(a,b) { return a+b }
// utilisation :
let c = somme(4,6);
```

Add a simple listener

This example demonstrates how to use `addEventListener()` to watch for mouse clicks on an element.

HTML

```
<table id="outside">
  <tr>
    <td id="t1">one</td>
  </tr>
  <tr>
    <td id="t2">two</td>
  </tr>
</table>
```

JavaScript

```
// Function to change the content of t2
function modifyText() {
  const t2 = document.getElementById("t2");
  const isNodeThree = t2.firstChild.nodeValue === "three";
  t2.firstChild.nodeValue = isNodeThree ? "two" : "three";
}

// Add event listener to table
const el = document.getElementById("outside");
el.addEventListener("click", modifyText, false);
```

In this code, modifyText() is a listener for click events registered using addEventListener(). A click anywhere in the table bubbles up to the handler and runs modifyText().

click est un **évenement** : peut être **mouseover, mouseout, mousemove**...

Appel de 2 (ou +) fonctions sur un évènement unique :

```
let bts=document.getElementById('butsub')
bts.addEventListener("click", () => {visibleOrNot(); imgFond()}, false);
```

Ici, visibleOrNot() et imgFond() ont été implémentées ailleurs. Butsup est l'ID d'un bouton.

Création de 2 fonctions pour alias de document.getElementById et Style :

```
function O(i){
    return typeof i =='object' ? i : document.getElementById(i);
  }

  function S(i){
    return O(i).style;
  }
```

Si i est un objet, on retourne simplement l'objet. Si c'est une variable, on retourne un résultat sous la forme document.getElementById()

exemple d'implémentation :

Html

```
<img id='img_obj' src='image.jpg'>
```

JavaScript

```
<script>
O('img_obj').onmouseover = function() { this.src='another_image.jpg' }
O('img_obj').onmouseout = function() { this.src='image.jpg' }
</script>
```

function() : fonction anonyme
this : indique à JavaScript d'opérer sur l'objet appelant.

Implémentation de S() :

Html

```
<body>corps avec uniquement ce texte.</body>
```

JavaScript

```
<script>
newdiv = document.createElement('div');
newdiv.id = 'maDivId';
document.body.appendChild(newdiv);
S(newdiv).border='solid 1px red';
newdiv.innerHTML = "je suis nouveau !!!" ;
</script>
```

Interroger un élément de style, avec getComputedStyle() :

```
<div class="classeTest" style="background-color: lightblue;">Ceci est un exemple de div</div>

<button onclick="afficherCouleur()">Afficher couleur</button>

<script>

function afficherCouleur() {


// Sélectionner l'élément div avec la classe "classeTest"

var element = document.querySelector('.classeTest');

// Obtenir les styles appliqués à l'élément

var styles = window.getComputedStyle(element);

// Extraire la couleur de fond

var couleurFond = styles.backgroundColor;

// Afficher la couleur dans la console

console.log("La couleur de fond de l'élément est : " + couleurFond); }

</script>
```

Modification d'un attribut Css :

```
document.body.style.backgroundImage = "url('img_tree.png')";
```

Créer et insérer des éléments

```
var newLink = document.createElement('a');
```

Affectation des attributs

```
newLink.id    = 'sdz_link';
newLink.href  = 'http://www.siteduzero.com';
newLink.title = 'Découvrez le Site du Zéro !';
newLink.setAttribute('tabindex', '10');
```

Insertion de l'élément

On utilise la méthode appendChild() pour insérer l'élément. Append child signifie « ajouter un enfant », ce qui signifie qu'il nous faut connaître l'élément auquel on va ajouter l'élément créé. Considérons donc le code suivant :

```
<!doctype html>
<html>
 <head>
  <meta charset="utf-8" />
  <title>Le titre de la page</title>
 </head>

 <body>
  <div>
   <p id="myP">Un peu de texte <a>et un lien</a></p>
  </div>
 </body>
</html>
```

On va ajouter notre élément <a> dans l'élément <p> portant l'ID myP. Pour ce faire, il suffit de récupérer cet élément, et d'ajouter notre élément <a> via appendChild() :

```
document.getElementById('myP').appendChild(newLink);
```

Ajouter des nœuds textuels

La méthode **createTextNode()** sert à créer un nœud textuel (de type #text) qu'il nous suffira d'ajouter à notre élément fraîchement inséré, comme ceci :

```
var newLinkText = document.createTextNode("Bla Bla Bla");

newLink.appendChild(newLinkText);
```

L'insertion se fait ici aussi avec appendChild(), sur l'élément newLink.

```
<body>
 <div>
   <p id="myP">Un peu de texte <a>et un lien</a></p>
 </div>

 <script>
   var newLink = document.createElement('a');

   newLink.id    = 'sdz_link';
   newLink.href  = 'http://www.siteduzero.com';
   newLink.title = 'Découvrez le Site du Zéro !';
   newLink.setAttribute('tabindex', '10');

   document.getElementById('myP').appendChild(newLink);

   var newLinkText = document.createTextNode("Le Site du Zéro");

   newLink.appendChild(newLinkText);
 </script>
</body>
```

Le fait d'insérer via appendChild() n'a aucune incidence sur l'ordre d'exécution des instructions. Cela veut donc dire que l'on peut travailler sur les éléments HTML et les nœuds textuels sans qu'ils soient au préalable insérés dans le document. Par exemple, on pourrait ordonner le code comme ceci :

```
var newLink = document.createElement('a');
var newLinkText = document.createTextNode("Le Site du Zéro");
newLink.id    = 'sdz_link';
newLink.href  = 'http://www.siteduzero.com';
newLink.title = 'Découvrez le Site du Zéro !';
newLink.setAttribute('tabindex', '10');
newLink.appendChild(newLinkText);
document.getElementById('myP').appendChild(newLink);
```

****

Ici, on commence par créer les deux éléments (le lien et le nœud de texte), puis on affecte les variables au lien et on lui ajoute le nœud textuel.
À ce stade, l'élément HTML contient le nœud textuel, mais cet élément n'est pas encore inséré dans le document :

La dernière instruction insère alors le tout.

RANDOM

```
function getRandomInt(min, max) {
// gestion d'une distribution aléatoire des images, min et max inclus (bornes)
    min = Math.ceil(min);
    max = Math.floor(max);
    return Math.floor(Math.random() * (max - min + 1)) + min;
  }
```

Foreach : *prend une fonction comme argument*

```
const a = ["a", "b", "c"];
a.forEach((element) => { console.log(element); });
```

event.preventDefault()

La méthode **`preventDefault()`**, rattachée à l'interface `Event`, indique à l'agent utilisateur que si l'évènement n'est pas explicitement géré, l'action par défaut ne devrait pas être exécutée comme elle l'est normalement.

ex :

Bloquer la gestion du clic par défaut

Lorsqu'on clique sur une case à cocher, par défaut, cela coche ou décoche la case. Dans cet exemple, on illustre comment bloquer ce comportement par défaut :

```
<document.querySelector("#id-checkbox").addEventListener(
  "click",
  function (event) {
    console.log("Désolé ! preventDefault() ne vous laissera pas cocher ceci.");
    event.preventDefault();
  },
  false,
);
```

event.preventDefault() <=> e.preventDefault()

BOUCLES

foreach

```
const array = [1, 5, 10];
array.forEach(function(currentValue) {
    console.log(currentValue);
});
```

Fonction en notation "arrow" :

```
const array = [1, 5, 10];
array.forEach(currentValue => console.log(currentValue));
```

PRATIQUE

history

```
window.history.back()          ► retour dans l'historique
window.history.forward()       ► en avant dans l'historique
```

```
function funcUrlBack(){
        let urlback = window.history.back()
        document.location.href = urlback
      }
```

getElementsByClassName() : attention : retourne une Collection.

Get all elements with class="example":

```
const collection = document.getElementsByClassName("example");
```

exemple de code :

```
const collection = document.getElementsByClassName("example");
for (let i = 0; i < collection.length; i++) {
  collection[i].style.backgroundColor = "red";
}
```

Vérifier l'existence d'une variable :

```
if (typeof variable == 'undefined') {
    document.write("La variable nbr n'existe pas <br/>");
} else {
    document.write("La variable nbr existe<br/>");
}
```

Ordonnancement / gestion du temps avec setTimeout, setInterval :

setTimeout : équivalent d'une tempo travail (on delay) en électrotechnique.
> exécute une fonction après un certain temps.

Exemple :

```
function bonjour(){ document.write("Bonjour !") }
setTimeout(bonjour, 2500)
```

fonction avec arguments :

```
function laFunction(prenom,nom){ alert("Bonjour " + prenom + " " + nom + ". Nous sommes le " + new Date().toLocaleString("fr-FR")) }
setTimeout(laFunction,1000,"Marcel","Proust") ;
```

Utilisation de fonctions fléchées :

```
setTimeout ( () => alert('Bonjour'), 1500 ) ;
```

Mettre fin à la fonction : clearTimeout

```
let t1 = setTimeout( () => alert('Bonjour.'), 1500 ) ;
document.write(t1) ;
clearTimeout(t1) ;
```

setInterval : même syntaxe que pour setTimeout
> appelle la fonction périodiquement, en se basant sur un intervalle de temps.

```
function laFonction(){ document.write("Bande de nazes.") }
setInterval (laFonction, 1500) ;
```

clearInterval :

```
var i=0;
var iddemo = document.getElementById("demo");

function laFonction(){
        iddemo.innerHTML += ("Bande de nazes. ");
        i+=1000;
        console.log("consolog", i);
        if (i>5000) { clearInterval(intervalle); document.write("Fini !"); }
}

let intervalle = setInterval (laFonction, 1500) ;
```

Open a new window and close the window after three seconds (3000 milliseconds):

```
const myWindow = window.open("", "", "width=200, height=100");
setTimeout(function() {myWindow.close()}, 3000);
```

Un exemple pratique de JavaScript sleep

```
function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

async function test() {
  console.log('test 1');
  await sleep(1000);
  console.log('test 2');
}
test();
```

Dans cet exemple, la fonction **test** affiche d'abord le message « test 1 », puis attend deux secondes (1000 millisecondes) grâce à la fonction **`sleep`**, avant d'afficher le message « test 2».

TIMER

```
let h = 0; let m = 0; let s=0;
var intervalle // variable globale

function timer(){
        intervalle = setInterval( function () {
                        p1.textContent = h+" : "+m+" : "+s;
                        s++;
                        if (s>59) { s=1; m+=1 }
                        if (m>59) { m=0; h+=1 }

                },1000)
}

function stopTimer(){
        clearInterval(intervalle);
}
```

HTML

```
<h2>---</h2>
<button type="button" id="butt" onclick="timer()">Go Timer</button>
<button type="button" id="butt2" onclick="stopTimer()">STOP Timer</button>
<p id="p1"></p>
```

Cacher / montrer les nœuds grâce à la propriété Element.classList et les méthodes add(), remove()

```
<button id="graphButt" type="button"><b>Graphe</b></button>
document.getElementById('graphButt').addEventListener("click", visibleOrNot, true);
```

```
var sfj = document.getElementById('selfreqJ'); // bouton radio "Jour"
var sfh = document.getElementById('selfreqH'); // bouton radio "Heure"
var sfh2 = document.getElementById('selfreqH2') // selection mutiple (heure)
var sfj2 = document.getElementById('selfreqJ2') // selection mutiple (jour)

function visibleOrNot (){

if (sfj.checked){
sfh2.classList.add('hidden');
sfj2.classList.remove('hidden');
}
else if (sfh.checked){
sfj2.classList.add('hidden');
sfh2.classList.remove('hidden');
}
```

hidden est une classe définie dans le css :

```
.hidden {
display: none;
/* width: 20%; */
}
```

Fonctions

```
// Defining the function:
function sum(num1, num2) {
  return num1 + num2;
}
// Calling the function:
sum(3, 6); // 9
```

```
// Named function
 function rocketToMars() { return 'BOOM!'; }
// Anonymous
 function const rocketToMars = function() { return 'BOOM!'; }
```

L'intérêt d'utiliser une fonction anonyme par rapport à une fonction nommée dépend du contexte et des besoins spécifiques du code. Quelques points à considérer :

- **Portée et Réutilisabilité** :
  - **Fonction nommée** : Les fonctions nommées peuvent être appelées avant leur définition grâce au hoisting (remontée) en JavaScript. Elles sont également plus faciles à réutiliser et à appeler récursivement, car elles ont un nom explicite.
  - **Fonction anonyme** : Les fonctions anonymes sont souvent utilisées comme arguments pour d'autres fonctions (par exemple, en tant que callbacks) ou pour créer des fonctions immédiatement invoquées (IIFE). Elles n'ont pas de nom, ce qui peut rendre le débogage plus difficile.
- **Lisibilité et Maintenance** :
  - **Fonction nommée** : Les fonctions nommées peuvent rendre le code plus lisible et plus facile à comprendre, car le nom de la fonction peut décrire son intention ou son rôle.
  - **Fonction anonyme** : Les fonctions anonymes peuvent rendre le code plus concis, mais elles peuvent aussi le rendre moins lisible, surtout si elles sont utilisées de manière excessive.

Callbacks : Les fonctions anonymes sont couramment utilisées comme callbacks pour des méthodes telles que `forEach`, `map`, `filter`, etc.

IIFE (Immediately Invoked Function Expression) : Les fonctions anonymes sont souvent utilisées pour créer des IIFE, qui permettent de créer une portée locale et d'éviter la pollution de l'espace de noms global.

Fonctions fléchées :

```
const sum = (param1, param2) => {
  return param1 + param2;
};
console.log(sum(2,5)); // => 7

const multiply = (a, b) => a * b;
console.log(multiply(2, 30)); // => 60
```

OBJETS

```
let jessica = new Object();

jessica = {
        nom: "Scroutdansmonpanier",
        prenom: "Jessica",
        age : 48
        }
```

On peut aussi définir ou accéder à des propriétés JavaScript en utilisant une notation avec les crochets.
Les objets sont parfois appelés « tableaux associatifs »

```
maVoiture["fabricant"] = "Ford";
maVoiture["modèle"] = "Mustang";
maVoiture["année"] = 1969;
```

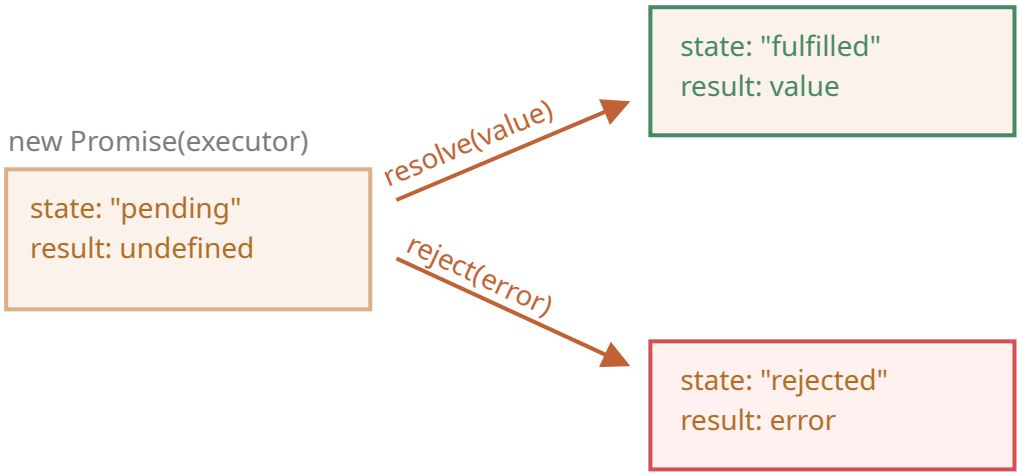
La valeur d'une propriété peut être une fonction, auquel cas la propriété peut être appelée « méthode »

```
let pascal = {name: "Pascal"}
let marie = {name: "Marie"}

function disBonjour(){
      alert("Bonjour " + this.name)
}

pascal.bonjour = disBonjour; // la fonction disBonjour() devient la propriété "bonjour" attachée à l'objet "pascal" (Méthode)
pascal.bonjour(); // affiche l'alerte "Bonjour Pascal"

function afficheProps(obj, nomObjet){
      let res = "";
      for (let prop in obj){
            if (obj.hasOwnProperty(prop)){ // pour chaque propriété attachée à l'objet
                  res+= `${nomObjet}.${prop}= ${obj[prop]}` + "<br>"
                  }
      }
      document.body.innerHTML += "<br>"+res;
}

afficheProps(marie, "marie");
afficheProps(pascal, "pascal");
```

Résultat (après l'alerte) :

> marie.name = Marie
>
> pascal.name = Pascal
> pascal.bonjour = function disBonjour(){ alert("Bonjour " + this.name) }

Fonctions asynchrones : PROMESSES / Promise




```
function loadScript(src){
        return new Promise((resolve, reject)=>{
        let s = document.createElement('script');
        s.src = src;
        document.head.appendChild(s);
        s.onload = () => resolve('Le fichier '+src+' a bien été chargé');
        s.onerror = () => reject(new Error ('Erreur : le fichier '+src+' n\'a pu être chargé.'));
        });

}

let promesse1 = loadScript('fic1.js');
let promesse2 = promesse1.then((result) => { alert(result) } , error => alert(error));
```

mots-clef (entités prédéfinies) : **Promise, resolve, reject, then, result, Error, error**

Une fonction sleep()

```
function sleep(ms){
return new Promise((resolve, reject) => {
        setTimeout(() => resolve(), ms);
        });
}
sleep(3000).then ( () => console.log('timeout fini !') );
```

Une "Promise" utilise souvent 2 fonctions, then et catch, **qui renvoient à leur tour une promesse**

```
const promesse = new Promise((resolve, reject) => { ... });
promesse.then (faireUnTruc);
        .then (faire1AutreTruc);
        .then (Faire Autre Chose);
        .catch (console.error);
```

Si une erreur se produit quelque part, il y a un appel du premier catch disponible dans la liste.

Le constructeur de Promise prend en paramètre une fonction.
Cette fonction possède 2 paramètres, qui sont eux-même des fonctions.

```
new Promise ((resolve, reject) => { ... });
```

resolve : fonction qui sera utilisée dans la promesse quand on aura réussi à compléter le traitement.
reject : sera déclenchée quand la promesse rencontre un problème.

ATTENTION :

L'instruction après .then doit aussi être une fonction (souvent, anonyme) sinon ça ne marche pas (sans pour autant déclencher d'erreur dans la console de déboggage JS)

Dans l'exemple de la fonction sleep(), on peut bien sûr enchaîner les .then :

```
sleep(1000).then( () => sleep(1000)).then( () => console.log("2s + tard'));
```

On peut aussi réaliser une fonction sleep sans promesse (fonction synchrone, bloquante) comme ceci :

```
function sleep2(ms) {
      const t = Date.now()
      while (Date.now() - t < ms) { } // on ne fait rien
}
```

Attention : consomme du CPU

IMPORTANT :

Dans un script, il peut y avoir ce que l'on pourrait appeler le **fil principal** (code synchrone), et le **fil secondaire** (asynchrone)

Le fil principal est traité en premier, et la partie asynchrone (tout ce qui se trouve après le mot-clé "then") sera exécuté ensuite.

Par exemple :

```
function sleep(ms){
    const t=Date.now();
    while ( Date.now() - t < ms ) {/* pause */}
}

const p1 = new Promise ( (resolve) => resolve(console.log("promesse p1")) );
p1.then ( () => console.log("p1 then") );
sleep(2000);
console.log("TEST");
```

résultat (dans l'ordre chronologique) :

promesse p1
TEST
p1 then

promesse p1 s'affiche instantanément, suivi, 2s plus tard, des 2 autres lignes
=> seule la ligne p1.then... est traitée de manière asynchrone et fait partie du fil secondaire, traité à la fin.

Autre écriture possible (plus propre) :

```
 function sleep(ms){
      return new Promise((resolve, reject) => {
      setTimeout(() => resolve(), ms);
      });
}
```

```
async wait2s() => {
      await sleep(1000);
      await sleep(1000);
      console.log('2s + tard');
}
```

On utilise ici le mot-clef await, en combinaison avec le paramètre async qui crée un "typage" asynchrone.

Encore mieux, on ajoute des **try / catch** :

```
async wait2s() => {
      try {
            await sleep(1000);
            await sleep(1000);
            console.log('2s + tard');
      }
      catch (err) {
            console.error(err);
            }
}
```

Une promesse peut se retrouver dans 3 états différents :

- pending => en cours
- fulfilled => complété positivement
- rejected => rejeté

```
function sleep(ms){
      return new Promise((resolve, reject) => {
            setTimeout(() => resolve("sleep terminé"), ms);
      });
}
const maRequete = new Promise ((resolve, reject) => {
      let random = Math.random().toFixed(2);
      if (random > 0.5){ resolve(random) }
      else { reject (new Error('trop petit'))}
});
maRequete
.then( (valeur1) => console.log("valeur random: ", valeur1) ) /* 0.76 */
.then(() => sleep(2000))
.then( (valeur2) => console.log("then sleep 2000 - ", valeur2) ) /* "sleep terminé" */
.catch(err => console.error(err));
```

Promise.all

permet de grouper des promesses (une sorte de collection de promesses), et on va pouvoir traiter la résolution ou non de l'ensemble de ces promesses. On récupère un tableau de valeurs correspondant au retour (resolve) de chaque promesse. Si au moins une erreur rencontrée : on passe au catch (échec)

exemple :

```
let x=2; let y=3;

const promise1 = new Promise( (resolve, reject) => {
    res=x+y;
    if (!res) { reject(new Error("erreur promise1!")) }
    resolve(res)
});
const promise2 = new Promise( (resolve, reject) => {
    res=x-y;
    if (!res) { reject(new Error("erreur promise2!")) }
    resolve(res)
});
const promise3 = new Promise( (resolve, reject) => {
    resolve(x*y)
});

Promise.all([promise1, promise2, promise3])
.then ( (values) => { console.log(values) } )
.catch(err => console.error("Il y a eu une erreur : ",err));
```

Résultat :

[5, -1, 6]

Promise.any

Fonctionnement un peu similaire, mais renvoie le résultat de la première promesse qui n'a pas échouée.
(et donc la première résolue => resolve)

Promise.allSettled()

renvoie les résultats sous la forme d'un tableau d'objets, avec le statut du retour (fulfilled, rejected) et la valeur retournée.
La fonction va ignorer les promesses qui vont échouer.

```
let x=2; let y=3;

const promise1 = new Promise( (resolve, reject) => {
    res=x+y;
    if (!res) { reject(new Error("erreur promise1!")) }
      resolve(res)
});
const promise2 = new Promise( (resolve, reject) => {
    res=x-y;
    if (res) { reject(new Error("erreur promise2!")) } // déclenche un statut "rejected"
    resolve(res)
});

Promise.allSettled([promise1, promise2])
.then ( (values) => { console.log(values) } )
.catch(err => console.error("Il y a eu une erreur : ",err)); // ne sera pas déclenché
```

Promise.race()

Parmi plusieurs promise, renvoie le résultat de la promesse qui se termine en premier : si c'est résolu (resolve) => ok, sinon, erreur (reject).

CLOSURES

My Def :
Fonctions à l'intérieur d'une fonction qui permet de retourner une variable "mémorisée", un peu comme s'il s'agissait d'une variable statique associée à la propriété d'une classe.

C'est une fonction qui a accès aux variables de son environnement lexical, même lorsque cette fonction est exécutée en dehors de cet environnement. En d'autres termes, une closure permet à une fonction de "se souvenir" de l'environnement dans lequel elle a été créée, y compris les variables locales qui étaient en portée à ce moment-là.

```
let compteur = function compte() {
        let cpt=0; // variable locale à la fonction
        return function() { // la fonction retournée a accès à la variable 'cpt'
                return cpt++
        }
}
```

Ici, la variable cpt se comporte comme une variable globale, et peut être associé à une instance de compteur :

```
a=compteur();
b=compteur();

console.log( a() ); // retourne 0
console.log( a() ); // retourne 1
console.log( b() ); // retourne 0
console.log( b() ); // retourne 1
console.log( a() ); // retourne 2
```
...

C'est une fonction qui retourne une autre fonction, et toute variable contenue dans cette dernière aura une portée globale.

map

Manipulation de nombres à l'aide d'une fonction fléchée :

```
const a1 = Array(4).fill(1).map((x,i) => (x+2*i)).join('.');
// 1.3.5.7
```

Manipulation d'un objet :

```
const persons = [
  {firstname : "Pierre", lastname: "Richard"},
  {firstname : "Raymond", lastname: "Poulidor"},
  {firstname : "Rika", lastname: "Zaraï"}
];

persons.map(getFullName);

function getFullName(item) {
  return [item.firstname,item.lastname].join(" ");
}
```

Génération du contenu HTML :

```
taskList.innerHTML = tasks.map(task => `
    <div class="task-item">
        <h3>${task.title}</h3>
        <p>${task.description}</p>
    </div>
`).join('');
```

- **`tasks.map(...)`** : La méthode `map` est utilisée pour transformer chaque objet `task` du tableau `tasks` en une chaîne de caractères HTML.
- **Template Literal** : Chaque tâche est transformée en un bloc HTML utilisant des template literals (délimités par des backticks `` ` ``). Chaque bloc HTML contient :
    - Un `div` avec la classe `task-item`.
    - Un titre `h3` affichant le `title` de la tâche.
    - Un paragraphe `p` affichant la `description` de la tâche.
- **`.join('')`** : La méthode `join` est utilisée pour concaténer toutes les chaînes de caractères générées par `map` en une seule chaîne HTML.