JAVASCRIPT INITIATION

Introduction

Pour l'historique voici le lien wikipédia : https://fr.wikipedia.org/wiki/JavaScript

Et maintenant entrons dans le vif du sujet : que peut-on concrètement faire avec ce langage et comment l'utiliser?
Vous avez certainement lu sur la page wikipédia que l'on pouvait travailler côté serveur et côté front, en ce ui concerne cette initiation nous allons voir la partie front, autrement dit, comment rajouter du dynamisme et de l'ergonomie à une page web avec l'aide de Javascript.

Comme il s'agit d'une initiation ce document est un résumé, je passerai volontairement sur certaines choses.
Pour plus de détailles je met des liens vers des explications plus poussées, et en fin de document se trouve des recommandations pour des livres et d'autres liens.

Explication sur les normes ECMAScript :
Pour résumer, les navigateurs possèdent en interne un moteur qui interprète et exécute le code Javascript.
Pour permettre le fonctionnement de ces moteurs ainsi que le développement d'applications (plugin, framework, code, etc...), il existe un ensemble de règles que l'on appelle norme ECMAScript (cette norme permet de standardiser l'utilisation d'un certain nombre de langages de programmation de type Script tels que le JS mais aussi l'Action Script). Sans cette norme, le code produit d'un site, par exemple, a peu de chance d'être compatible sur tout les navigateurs.
Suivant les années, il existe plusieurs éditions de cette norme.

Pour les besoins de cette initiation nous allons travailler avec la norme ECMAScript5 ou ES5.
Lien vers l'explication de ces normes : https://fr.wikipedia.org/wiki/ECMAScript .

Pour commencer où placer le code :
Il y a plusieurs manières de gérer du code dans une page html.
Premièrement, directement dans la page html, dans la balise <head></head> il est possible de mettre du code entre les balises.

Exemple:

```
<script type="text/javascript">
        mon code JS
</script>
```

Ce type de d'inclusion est généralement utilisé pour de très petits bouts de code comme pour démarrer un plugin (un script de galerie et autre...)

Une autre façon d'inclure du script est de charger un fichier externe en le liant à la page (de la même manière qu'on lie un fichier Css) :

Exemple:
*<script type="text/javascript" **src="js/main.js"**></script>*

Le code se trouve dans le fichier main.js, fichier qui se trouve dans le dossier [js] de votre site.
Le paramètre "src" signifie "source".
Pour rappel, ce type de construction est le même que pour la balise "img".

A noter qu'il est possible (et souvent pratiqué) de mettre cette balise script en bas de la page avant la fermeture de la balise body pour des raison de chargement.
En effet, les scripts étant chargés de manière synchrone, cela signifie que le navigateur attend que le fichier soit téléchargé pour passer à la suite, aussi si on a une page demandant beaucoup de ressources externes, de médias, de node html, de styles à charger et à interpréter, il est préférable de localiser les scripts secondaires en bas de page car, le navigateur interprétant le contenu de haut en bas, ceux-ci seront chargé et "lus" à la fin et ne ralentiront donc pas l'affichage de la structure et des éléments essentiels.

Les commentaires :
Comme pour tout langage, on peut (on doit !!) commenter son code.
Il existe deux sortes de commentaires : le commentaire en ligne // et le commentaire en bloc /**/.

Le commentaire en ligne, comme son nom l'indique, permet de commenter une ligne, et, lors d'un retour à la ligne, de sortir de son action.

Exemple:
*// **-début-** un commentaire sur une ligne... **-fin-***
Cette ligne n'est plus commentée.....

Le commentaire de bloc permet de commenter tout un pan de code ou de description, il a un ouvrant /* et un fermant */ (comme en css)

Exemple:
***/* -début-** un bloc de commentaire*
un bloc de commentaire
*un bloc de commentaire **-fin- */***
Cette ligne n'est plus commentée.....

Dans la pratique, vous devrez souvent revenir sur votre code soit pour l'optimiser, soit pour corriger des bugs ou encore pour collaborer à un projet. Commenter est essentiel pour la clarté et la productivité du code.

Dialoguer avec le navigateur :
Lors du développement de vos scripts Front, vous aurez à tester de façon régulière votre code, afin de corriger une erreur ou de connaître le résultat d'une opération, le contenu d'une variable et autre...
Pour ce faire, il est possible d'ouvrir la console (raccourci [F12]) et d'y afficher des éléments envoyés par votre script en utilisant les méthodes console.log(), console.error(), console.warn() et console.table() (il y en a d'autres mais celles-ci sont les plus courantes).

Toutes ses méthodes possèdent leurs différences :
console.log() permet d'afficher texte, chiffre, de dérouler des objets, etc...

Exemple :
var bool = true;
console.log(*bool) // la console affiche true (la valeur stockée dans la variable)*

console.warn() permet d'afficher le même type de contenu mais formaté en avertissement, un panneau exclamation devant le résultat sur fond jaune.
console.error() permet d'afficher le même type de contenu mais formaté en erreur, une bulle rouge avec croix devant le résultat sur fond rouge.
console.table() permet d'afficher des tableaux ou des objets littéraux mais formatés avec "index" d'un côté et "value" de l'autre, sous forme de tableau. Non compatible avec windows edge.
Attention à toujours enlever les consoles du code lors de la livraison car cela consomme de la ressource (et en laisser n'est pas professionnel).

Bonne pratique : l'indentation (ou retrait de ligne) :
Quand une structure de code est imbriquée dans une autre, il est important d'utiliser l'indentation pour une meilleur lecture du code.
En effet, l'indentation permet de mieux voir si quelque chose ne va pas ou simplement de s'y retrouver plus facilement dans un ensemble de blocs d'instructions.
Il arrive souvent, au début, d'oublier une parenthèse (), une accolade {} ou un crochet [], l'indentation permet de voir qui dépend de qui et de corriger le tir.

Exemple illisible :

```
var tab = ["m","v","c"];
function test(paramTab){
var nb = paramTab.length;
for (var i=0;i<nb;i++){
if(paramTab[i]=="m"){
console.log(paramTab[i]);
}else{
console.log("pas de m dans le tableau")
}
}
test(tab);
// la console affiche Uncaught ReferenceError: test is not defined, en effet, il y a une erreur mais sans l'indentation il est très difficile de la voir.
```

Exemple lisible :

```
var tab = ["m","v","c"];
function test(paramTab){
    var nb = paramTab.length;
    for (var i=0;i<nb;i++){
        if(paramTab[i]=="m"){
            console.log(paramTab[i]);
        }else{
            console.log("pas de m dans le tableau")
        }
```

```
        } // il manquait cette accolade fermante
    }
    test(tab);
```

Les indentations permettent, tout comme pour le html et le css, de voir quel bloc est enfermé dans quel autre (on parle de profondeur) : la variable "nb", la boucle "for" et la condition "if" appartiennent à la "function", les "consoles" appartiennent à la condition "if".
Le raccourci d'indentation est [tab].

Les variables

Déclaration :

Une variable est un espace de stockage qui permet de garder en mémoire de l'information pour ensuite la réutiliser.
Pour déclarer et utiliser une variable, il faut respecter quelques règles :
Tout d'abord, on déclare avec le préfixe "var" puis le nom de la variable ; il n'est pas obligatoire de donner une valeur à la variable dés le début, si on le fait, on parle alors d'initialisation.

Exemple:
var maVariable; *ou* ***var maVariable = 0;***
puis dans le reste de votre code, pour utiliser la variable il suffit de la réécrire sans le préfixe var.

```
function test(){
        maVariable = 20;
}
```

Le Javascript étant sensible à la casse, il faudra toujours écrire le nom de la variable de la même façon, sinon on risque de provoquer une erreur ou un doublon :

Exemple:
var maVar *et différent de* ***var MaVar*** *ou de* ***var mavar***

Il est interdit de commencer le nommage par un chiffre, mais on peut finir avec.
Il est interdit d'utiliser certains caractères spéciaux dans le nom de la variable comme #@- et autres.
Il est interdit d'utiliser un mot clef (réservé) du langage comme nom de variable :

Exemple:
var in = 0; var catch = "test"; *ne sont pas corrects car "in" et "catch" sont des mots "réservés".*

Il est TRES fortement recommandé d'utiliser un nommage logique : une variable qui contient des informations de connexion s'appellera logiquement userPass ou userName.
Par convention, on utilise le cammelCase pour les noms long : on commence par une minuscule puis une majuscule à chaque nouveaux mots composant le nom (par convention, les majuscules en premier caractère sont réservées aux déclarations d'objets ou de classes, mais on y reviendra).

les types :

Dans la plupart des langages évolués il est obligatoire de déclarer le type d'information (texte, nombre ou autre) que l'on stocke dans une variable.

Ce n'est pas le cas en Javascript, la variable prend de manière dynamique le type d'information qui s'y trouve, on dit donc que le Javascript est un langage à typage faible.

Exemple:

***var maVar** = **10**; chiffre donc type number*

***var monAutreVar** = **"chien"**; texte donc type string.*

Pour tester le type d'une valeur, Javascript possède l'opérateur "typeof" .
Il en existe 6 (en fait 7 avec ES6) :

- Booléen → true / false.
  La valeur d'un booléen ne peut être que "true" ou "false", soit vrai soit faux, c'est une valeur d'opération logique.

- Null
  Javascript renvoie "null" lorsqu'une opération a été demandée, mais que rien n'a été trouvé, c'est une absence de valeur mais pas du vide.

  Exemple:

  ***var string** = **"test"**; // création d'une variable et assignation (=) d'une valeur chaîne de caractère (donc type string).*

  ***var result** = **string**.match(/v/g)*

  *console.log(**result**); // on recherche avec la méthode "match" s'il y a un caractère "v" dans la chaîne, la console du navigateur affiche le contenu de la variable **result** après l'opération effectuée par "match" : rien n'est trouvé donc = null;*

- Undefined → non défini.
  Par exemple, une variable utilisée mais jamais créée ou une variable sans valeur est de nature "undefined" :

  Exemple:

  var maVar;

  *console.log(**maVar**); // renvoi undefined dans la console navigateur.*

  *console.log(**monAutreVar**); // renvoi "Uncaught ReferenceError: monAutreVar is not defined"*

- Number
  Représente le type d'un chiffre, qu'il soit entier (int) ou à virgule (float).

  Exemple:

  ***var monNombre** = **10**;*

  *console.log(typeof **monNombre**); // le navigateur renvoie "number" (typeof est l'opérateur qui permet de demander le type d'une donnée).*

  ***var monNombre2** = **12,5**; // le navigateur renvoie "number".*

- String → chaîne de caractère

  Exemple:

  ***var maChaine** = **"blablabla"**;*

  *console.log(typeof **maChaine**) // le navigateur renvoie "string";*

- Object

Le type objet est particulier car il regroupe beaucoup de notions qui ont trait à la programmation orientée objet (POO). Pour faire simple, un objet est un ensemble de propriétés (variable interne) et méthodes (fonction interne);
Il existe toute sorte d'objets comme les objets littéraux, les objets natifs déjà créés dans le langage, tel que Array(), Image(), Math, Date(), etc.

La portée ou scope

La portée d'une variable est en quelque sorte son accessibilité à travers le code :
imaginons une variable que l'on déclare au début d'un script, cette variable étant attachée au contexte global, le script, elle est donc utilisable partout. J'entends par là qu'on peut l'utiliser dans tout les blocs de code et les fonctions du script.

Exemple:

```
var maVar = "test";
if(true){
    console.log(maVar); // renvoi test;
}
function maFonction(){
    console.log(maVar); // renvoi test;
}
maFonction();
```

Dans ce cas, on voit bien que la variable est visible dans le bloc condition et dans la fonction, elle est donc globale.

Exemple 2:

```
if(true){
    console.log(maVar); // renvoi Uncaught ReferenceError: maVar is not defined
}
function maFonction(){
    var maVar = "test";
    console.log(maVar); // renvoi test;
}
maFonction();
```

Dans ce cas, la variable est créée dans une fonction, cette fonction est donc le contexte de la variable, ce qui signifie que cette variable n'existe que dans la fonction, une fois que celle-ci a terminé son exécution, la variable est détruite.
La variable étant limitée à la fonction, elle n'a donc pas d'existence (de portée) en dehors et ne peut pas être utilisée dans d'autres blocs (depuis ES6 il y a d'autres manière de déclarer une variable ce qui influe sur la portée).
Précision : comme l'objet window est le contexte global lorsqu'on crée une variable globale dans le script, on crée en fait de manière dynamique une propriété sur l'objet window :

Exemple:

```
var aa =  "truc"; (en tout début de script et hors fonction)
console.log(window) // Après avoir déroulé l'objet window dans la console, on voit bien que sa première propriété (clef) est aa et qu'elle a pour valeur "truc".
```

Pour aller plus loin : https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/var

les principales boucles

Les boucles servent à répéter des actions ou traitements de manière à ne pas avoir à écrire beaucoup de lignes de code quand une action est demandée plusieurs fois.
Par exemple, si on souhaite écrire quatre fois bonjour, on lancera une boucle qui le fera quatre fois au lieu d'écrire ces quatre lignes de texte à la main.

Exemple: - (je décomposerai le code plus tard) -

```
for(var i = 0; i < 4; i++){
    console.log("bonjour "+(i+1)+"fois");
}
// la console affiche :
console.log("bonjour 1fois");
console.log("bonjour 2fois");
console.log("bonjour 3fois");
console.log("bonjour 4fois");
```

Les différentes type de boucles

- La boucle for :

elle répète ses instructions tant que sa condition est valide.

for(valeur de départ → **var i = 0;**condition→**i<4;**incrémentation→**i++){**

```
    bloc de code à exécuter.
    console.log("bonjour "+(i+1)+"fois");
}
```

Décomposition :

- Étape 1 : on commence par initialiser une variable "i" et lui assigner une valeur 0. Cette variable va nous servir à tester une condition. Cette instruction n'est exécutée qu'une fois au lancement de la boucle.

Étape 2 : on met en place une condition qui va nous servir à arrêter la boucle à la fin de son traitement. Si "i" est inférieur (<) à 4 alors …

Étape 3 : … on rentre dans le corps de la boucle { et } traitement du bloc de code qui s'y trouve.

Etape 4 : la variable "i" est incrémentée : 1 est ajouté à sa valeur.

Retour à l'étape 2, re-test si "i" est inférieur à 4, puis les étapes suivantes sont de nouveau accomplies jusqu'à ce que la condition de la 2ème étape ne soit plus remplie (à force d'incrémentation, la valeur de "i" passe à 4 donc égal pas inférieur).

- La boucle while:

Comme la boucle "for" cette boucle a besoin d'une condition pour fonctionner.

while(condition**){**

```
    bloc de code à exécuter.
}
```

La condition est testée, si "true" alors exécution des instructions sinon, arrêt.

Exemple:

var tab = ["truc","chose","machin"]; (création d'un tableau simple pour l'exemple)

```
console.log(tab); // affiche ["truc","chose","machin"].
while(tab.length > 0){// length correspond à la longueur du tableau.
        tab.pop();  (la méthode pop permet de retirer une valeur à la fin d'un tableau)
        console.log(tab);
}
la console donne successivement :
["truc","chose"]
["truc"]
[]
puis s'arrête.
```

Au début de la boucle la condition est respectée (renvoie true) car le tableau a bien trois valeurs de stockées.
Puis, à l'aide de la méthode "pop()" du tableau, on supprime une valeur, on affiche la console puis on revient au début de la boucle jusqu'à ce que la condition de la boucle ne soit plus respectée et renvoie "false" (nombre de valeur dans le tableau égale à 0).

- La boucle do while :

Parente de la boucle "while", la boucle "do while" exécute des instructions puis teste une condition. Si cette condition est respectée la boucle exécute de nouveau les instructions etc, etc, jusqu'à ce que la condition ne soit plus valide.
La grande différence c'est que les instructions sont exécutées au moins une fois.

```
do{
    bloc de code a exécuter.
}while(condition);
```

- La boucle for in:

Cette boucle s'utilise sur des objets qui possèdent des propriétés énumérables.

```
for(variable in objet){
    bloc de code a exécuter.
}
```

La condition de cette boucle est : tant qu'il y a des propriétés dans l'objet exécute les instructions.

```
Exemple:
// création d'un objet littéral avec des paires clef:valeur.
var obj = {
        prop1:"truc",
        prop2:"chose",
        prop3:"machin"
}
for(var key in obj){
        console.log(key); // affiche le nom de la propriété stockée dans la variable
        console.log(obj[key]);// affiche la valeur correspondant à la propriété dans l'objet.
}
la console donne successivement :
prop1
truc
prop2
```

chose
prop3
machin
puis s'arrête : il n'y a plus de propriétés à énumérer dans l'objet.

Casser une boucle

L'instruction "break" permet d'interrompre une boucle en cours de traitement.
Imaginons que l'on ait un tableau contenant des mots de passe ou des noms mais sans connaître leurs disposition dans le tableau.
On souhaite tester si le nom ou le mot de passe correspondent à une donnée que l'on possède mais en stoppant le traitement si l'on trouve une correspondance avant la fin de la boucle.
Break permet alors d'interrompre la boucle avant sa fin

Exemple:

```
var data = "machin"; // la donnée recherchée
var tabData = ["truc","chose","machin","bidule","vgghajjkb"];// le tableau dans lequel on cherche
// i démarre à 0, tant que i est inférieur à la longueur du tableau; i + 1
for(var i = 0; i < tabData.length; i++){
        console.log(tabData[i]);// affiche la donnée du tableau a l'index de i
        if(data === tabData[i]){// si le contenu de data correspond au contenu du tableau a cet index :
                console.log("sortie de la boucle"); // affichage
                break; // sortie de la boucle (donc on ne lit pas le reste du tableau)
        }
}
```

La console affiche :
truc
chose
machin
sortie de la boucle
puis s'arrête, l'instruction break a cassé la progression de la boucle, le reste du tableau n'est pas traité.

Pour en savoir plus https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Boucles_et_it%C3%A9ration

les opérateurs

Je ne parlerai ici que des opérateurs les plus courants et utilisés, afin aller plus loin (après initiation) je mets un lien en fin de description.

Opérateur d'affectation :

Un opérateur d'affectation sert à donner une valeur, donc affecter, à une variable. Cette valeur peut être de tout type.
Le premier, affectation simple est "=".

Exemple :

```
var test = "ici";
```

ou

var n = 10;

affectation après addition +=,

affectation après soustraction -=,

affectation après multiplication *=,

affectation après division /=,

affectation du reste de division %=.

Voyez ce type d'affectation comme variable = variable + 10.

var test = 10;

Exemple de += :

test += 12;

console.log(test); // donne 22, donc 10 est la valeur de la variable + 12, on peut dire que la variable est égale à elle-même (la valeur stockée précédemment) plus 12.

Exemple de -= :

test -= 6;

console.log(test); // donne 4.

*Exemple de *= :*

*test *= 5;*

console.log(test); // donne 50.

Exemple de /= :

test /= 2;

console.log(test); // donne 5.

Exemple de %= : (dans ce cas précis % signifie modulo : le reste d'une division)

test %= 3;

console.log(test); // donne 1;

Opérateur de comparaison :

Les opérateurs de comparaison permettent de comparer deux valeurs et de déterminer s'il y a égalité ou différence : si la comparaison renvoie vrai (true) ou faux (false). Ces opérateurs fonctionnent avec des chiffres mais aussi du texte, des booléens, etc...

- Opérateur d'égalité ==,
- Opérateur d'égalité stricte === vérifie les valeurs ainsi que leurs types,
- Opérateur d'inégalité != (différent de),
- Opérateur d'inégalité stricte !== vérifie si les valeurs sont inégales et si leurs types sont inégaux,
- Opérateur supérieur à > vérifie si la première valeur est supérieure à la deuxième,
- Opérateur supérieur ou égal à >= vérifie si la première valeur est supérieure ou égale à la deuxième,
- Opérateur inférieur à < vérifie si la première valeur est inférieure à la deuxième,
- Opérateur inférieur ou égal à <= vérifie si la première valeur est inférieure ou égale à la deuxième.

Exemple == :

console.log(15 == 15) // renvoie true, !Attention 15 == "15" renvoie true car cet opérateur ne vérifie pas le type ;

Exemple === :

console.log(13 === 13) // renvoie true, 13 === "13" renvoie false car cet opérateur vérifie le type. On compare 13 en chiffre et "13" en caractère donc faux.

Exemple != :
console.log(150 != "truc") // renvoie true, les deux valeurs sont différentes.
console.log("truc" != "truc") // renvoie false, les deux valeurs ne sont pas différentes.

Exemple !== :
console.log("bidule" !== "truc") // renvoie true, les deux valeurs sont différentes.
console.log(22 !== 22) // renvoie false, les deux valeurs ne sont pas différentes et pas du même type.

Exemple > :
console.log(3 > 1) // renvoie true.
console.log(3 > 4) // renvoie false.

Exemple >= :
console.log(12 >= 5) // renvoie true.
console.log(7 >= 7) // renvoie true.

Exemple < :
console.log(12 < 5) // renvoie false.
console.log(5 < 7) // renvoie true.

Exemple <= :
console.log(12 <= 12) // renvoie true.
console.log(7 <= 10) // renvoie false.

Opérateurs mathématiques :

Ce sont les opérateurs classiques qui permettent de faire des additions +, des soustractions -, des multiplications *, des divisions / et modulo %.
Il est aussi possible d'utiliser l'opérateur d'incrémentation ++, l'opérateur de décrémentation --, et l'opérateur de négation -.

Exemple ++ :
var a = 3;
console.log(a++); // renvoie 3
console.log(a) // renvoie 4
Explication : lors de l'incrémentation, en mettant le ++ après la variable, on a d'abord affiché la valeur de la variable puis fait l'opération a=a+1, c'est pourquoi la deuxième console affiche le résultat de l'opération.
console.log(++a); // renvoi 4 car l'opération ++ a été effectuée avant l'affichage.

même chose pour -- mais en négatif

Exemple - :
var b = 9;
console.log(-b) // renvoie -9, l'exact opposé de la valeur initiale, cela revient à convertir en négatif en multipliant par moins un (utile dans de rare cas) 9-1.*

Opérateurs logiques :

Les opérateurs logiques sont au nombre de trois : ET(AND) &&, OU(OR) ||, NON !.
Ils permettent de tester des valeurs booléennes.
Construction de AND (valeur **&&** valeur) // renvoie true ou false.

Exemple 1:
var num = 10;
console.log(num>5 ***&&*** *num<15) // renvoie true;*
// num(10) est supérieur à 5 AND num(10) est inférieur à 15 donc javascript voit notre code comme suit :
console.log(true ***&&*** *true) le résultat est donc "true".*

Exemple 2 :
console.log(num>11 ***&&*** *num<15) // renvoie false;*
// num(10) n'est pas supérieur à 11 AND num(10) est inférieur à 15 donc javascript voit notre code comme suit :
console.log(false ***&&*** *true) le résultat est donc "false".*

On peut le voir comme suit :
[true] **&&** [true] *// donne true.*
[true] **&&** [false] *// donne false.*
[false] **&&** [true] *// donne false.*
[false] **&&** [false] *// donne false.*

Si les deux valeurs sont "true" alors l'opérateur renverra "true" sinon, dans tout les autres cas, il renverra "false" (un petit dernier pour la route : console.log(num>5 && num<15 && num != 10) donc console.log(true && true && false) donc le résultat est "false").

Construction de OU (valeur || valeur) // renvoie true ou false

Exemple 1 :
var num = 25;
var nb = 13;
console.log(num == nb || num < 30); // renvoie true
// num(25) n'est pas égal à nb(13) OU num(25) est inférieur à 30
donc javascript voit notre code comme suit : console.log(false || true) le résultat est donc "true" car au moins une des deux valeurs est true.

Exemple 2 :
var num = 25;
var nb = 13;
console.log(num == nb || num != 25); // renvoie false
// num(25) n'est pas égal à nb(13) OU num(25) n'est pas différent de 25 donc javascript voit notre code comme suit : console.log(false || false) le résultat est donc "false" car les deux valeurs sont "false".

On peut le voir comme suit :
[false] || [true] *// donne true.*
[true] || [false] *// donne true.*
[true] || [true] *// donne true.*
[false] || [false] *// donne false.*

Si au moins une valeur est "true" alors l'opérateur renverra "true" sinon "false".

Construction de NON (**!**valeur), permet de gérer l'inverse d'un booléen :

Exemple 1:
var bool = false;
*console.log(**!**bool); // renvoie true;*
// NOT false(bool) donc donne true, l'évaluation avec l'opérateur NON donne l'inverse de la valeur de départ car un booléen ne pouvant être que "true" ou "false" le résultat est forcément contraire.
Javascript le voit comme suit (not false)

Exemple 2 :
var num = 25;
var nb = 13;
*console.log(**!**(num !== nb)); // renvoie false*
// NOT(num est différent de nb donc donne true) donc donne false.

Opérateurs de concaténation :

L'opérateur de concaténation + permet d'associer deux chaînes de caractère ensemble.

Exemple :
var unPrenom = "Jean";
console.log("Bonjour "+unPrenom+" bienvenue sur mon site.") // affichera dans la console "Bonjour Jean bienvenue sur mon site."

Dans cet exemple, on concatène une chaîne avec une valeur stockée dans une variable à une deuxième chaîne.

Pour aller plus loin :https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Expressions_et_Op%C3%A9rateurs

Structures conditionnelles :

Les structures conditionnelles sont des instructions qui permettent d'effectuer des tests "oui / non" (true / false) et de lancer des actions ou pas. Les résultats des tests effectués renvoient toujours un booléen true ou false. Pour effectuer ces tests, on se sert des opérateurs conditionnels et logiques vus plus haut.

Structure minimale (si) "if" :

```
if(condition test){
    bloc de code à exécuter.
}
```

Considérez cela comme : si le test demandé renvoie true alors on effectue les instructions contenues dans le corps de la condition sinon on ne fait rien.

Exemple 1:
var test = 3;
if(test < 10){
*   console.log("test est bien inférieur");*
}

```
// La variable est bien inférieure à 10, la condition est bien respectée, son résultat est "true", la console du navigateur affiche bien "test est bien inférieur".

Exemple 2:
var test = 3;
if(test > 10){
    console.log("test est bien supérieur");
}
// La variable n'est pas supérieure à 10, la condition n'est pas respectée, son résultat est "false", la console du navigateur n'affiche rien car on ne rentre pas dans le corps {} de la condition, on ne peut donc pas lancer le bloc de code.
```

Structure "if" "else" :

Cette structure n'est pas plus compliquée que la première, elle permet d'effectuer une action si la condition n'est pas valide.

```
if(condition test){
    bloc de code à exécuter.
}else{
    bloc de code à exécuter.
}
```

```
Exemple 1 :
var prenom1 = "Jean"; // petite remarque : pas d'accent dans le nom de la variable, le chiffre a la fin.
var prenom2 = "Pierre";
if(prenom1 != prenom2){ // la condition renvoie "true" , prenom1 est bien différent de prenom2
    console.log(prenom1+" est bien différent de "+prenom2);// au passage utilisation de la concaténation avec +.
}else{
    console.log(prenom1+" n'est pas différent de "+prenom2);
}
// La console affiche "Jean est bien différent de Pierre" car le test [valeur1 est différente de valeur2] renvoie "true", on entre bien dans le corps {} de "if", on ne rentre pas dans le corps {} du "else".

Exemple 2 :
var prenom1 = "Jean".
var prenom2 = "Pierre".
if(prenom1 == prenom2){ // la condition renvoie "false" , prenom1 est différent de prenom2 mais là avec l'opérateur "==" on demande une égalité.
    console.log(prenom1+" est bien pareil à "+prenom2);
}else{
    console.log(prenom1+" est différent de "+prenom2);
}
// La console affiche "Jean est différent de Pierre" car le test [valeur1 est égale à valeur2] de "if" renvoie "false", il n'est donc pas possible d'effectuer le premier bloc de code, ce qui nous fait rentrer dans le else.
```

Avec cette structure si le test de condition if renvoie "false", else est toujours valide.

Structure "if" "else if" :

```
if(condition test){
```

```
    bloc de code a exécuter.
}else if(autre condition test){
    bloc de code a exécuter.
}
```

Note : il est possible, à partir de là, de rajouter des "else if" ou un "else", c'est selon la série de tests à effectuer. Pour ce genre de cas de figure, nous verrons plus tard une autre structure conditionnelle plus appropriée.

```
Exemple 1:
var unType = "bidule";
if(typeof unType === number){
    console.log(unType+" est un nombre");
}else if(typeof unType === string){
    console.log(unType+" est une chaîne de caractère");
}
// Dans cet exemple on teste le type de la variable pour savoir si c'est un chiffre ou une chaîne, le type du contenu de la variable étant "string" (une chaîne), la première condition (if) renvoie "false", on passe à la deuxième (else if) qui renvoie "true", la console affiche donc "bidule est une chaîne de caractères".
```

Note : Il est naturellement possible de faire des structures imbriquées.

```
if(condition test){
    if(condition test){
        bloc de code a exécuter.
    }else{
        bloc de code a exécuter.
    }
}
```

```
Exemple :
var life = 100;
if(typeof  !== "undefined"){
    if(life > 50){
        console.log("il reste à votre héros plus de 50 pv");
    }else{
        console.log("attention vous êtes en dessous de 50pv");
    }
}
// Dans cet exemple on imbrique deux tests : vérification de la présence d'une valeur dans la variable en testant le type du contenu : s'il n'est pas typé "undefined" la condition renvoie "true", c'est donc que l'on peut effectuer un traitement, sinon renvoie "false", on ne passe pas au deuxième test et donc rien n'est fait. Un traitement effectué sur "undefined" renvoie une erreur bloquante (on ne fait pas de traitement sur du rien, ce type de test permet d'éviter cette erreur).
Le deuxième teste la valeur du "number" : s'il est supérieur à 50 la console affiche "il reste à votre héros plus de 50 pv", sinon "attention vous êtes en dessous de 50pv".
```

Structure conditionnelle avec opérateur logique :

Comme vu plus haut, les structures conditionnelles renvoient toujours un booléen, il est possible d'utiliser les opérateurs logiques dans ces tests :

```
if(condition test && condition test){
    bloc de code a exécuter.
}
```

Exemple:

```
var nom = "Jean";
var positionListe = 10;
if(nom === "Jean" && positionListe < 20){
    console.log("votre prénom est "+nom+" et votre position dans la liste est "+positionListe)
}
// Javascript le voit comme suit : (true and true) donc true.
```

Test absolument inutile mais qui démontre l'utilisation d'un double test afin d'effectuer une action.

Une autre structure conditionnelle : le switch case :

Le switch case permet de tester une condition selon plusieurs valeurs :

```
switch(valeur){
  case cas 1:
    bloc de code a exécuter.
    break;
  case cas 2:
    bloc de code a exécuter.
    break;
  default:
    bloc de code a exécuter.
}
```

"switch" évalue la valeur de départ donnée dans les parenthèses avec chacune des données se trouvant en face des "case", si une correspondance est trouvée alors le bloc d'instruction se trouvant directement après est exécuté puis break qui stoppe l'évaluation, fin du switch.

"Défault" : facultatif, permet d'exécuter du code si aucune évaluation "case" n'a abouti à un résultat.

Exemple :

```
var main = "pierre";
switch(main){
    case "ciseau" :
        console.log("Gagné !");
        break;
    case "pierre" :
        console.log("Ex æquo, recommence !");
        break;
    case "feuille" :
        console.log("Perdu !");
        break;
    default:
        console.log(" Arrête avec le puits, ça n'existe pas !");
}
// dans un premier temps la valeur de main(pierre) va être évaluée avec le premier "case"(ciseau), pas de correspondance donc pas de lancement de la console ni de son break.
```

Évaluation avec le deuxième "case"(pierre), une correspondance est trouvée, la console affiche "Ex æquo, recommence !", le break sort du switch.
Le troisième "case" n'est pas évalué et le "default" n'est pas lancé.

Il est possible d'enchaîner plusieurs "case" ensemble de façon à faire une même action pour plusieurs évaluations.

```
Exemple :
var type = typeof "meuh!";
switch(type){
case "number" :
case "string" :
    console.log("Le type "+type+" est différent d'object");
    break;
case "object" :
    console.log("Le type est "+type);
    break;
}
// Cette construction à double case est valide.

// note : je vous laisse déterminer le résultat dans la console, au passage si vous changez la valeur testée typeof par null (qui, je vous le rappelle est défini comme un type à part entière, vous allez avoir une surprise..., je vous laisse le plaisir de chercher la réponse dans la doc sur internet).
```

Pour aller plus loin sur les conditions :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/if...else

Pour aller plus loin sur le switch case :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/switch

les fonctions

Une fonction est un bloc d'instructions très important en javacript, qui peut être appelé par le script global, dans d'autres fonction et blocs de code, par elle-même sous certaines conditions (on parle dans ce cas, de fonction récursive).
Elle peut être anonyme, prendre des paramètres et rendre des résultats après traitement...

Mais commençons par le début : basiquement une fonction s'écrit comme suit :

```
function nomDeLaFonction(){
    bloc de code a exécuter.
}
nomDeLaFonction();
```

Pour déclarer une fonction, on utilise le préfixe "function" puis un nom, des parenthèses (qui peuvent être vides) et des accolades ouvrantes {et fermantes} qui représentent le corps de la fonction dans laquelle se trouve le code a exécuter.

Première remarque :
le nom d'une fonction suit les mêmes règles que pour celui d'une variable : pas de caractères spéciaux, pas de chiffres au début, utilisation du camelCase pour la compréhension et un nom logique, si l'on crée une fonction qui recherche en série le prix d'un certain nombre d'articles d'un panier client, autant appeler logiquement la fonction "getPricesOfCart()" (par convention très souvent les noms sont écrit en anglais cela dépend principalement de l'équipe).
Pour voir pourquoi les fonctions sont fondamentales, il faut comprendre que, lorsque du code est mis dans le script comme une condition ou une boucle par exemple, celui-ci est exécuté dés le lancement de la page.
Premier point : jusqu'à maintenant, tous les exemples donnés se lancent dés le chargement du script (de la page donc), hors pour la plupart des traitements à effectuer, comme l'ouverture d'un menu, la vérification et l'envoi des datas d'un formulaire ou l'ouverture d'un popup, il faut attendre soit une action de l'utilisateur soit le téléchargement de ressources (image, autre script...) etc.
Les fonctions, après création, doivent être appelées par leurs noms pour être exécutées, vous pouvez donc les appeler lors d'un clic, à la fin du chargement d'une galerie, à l'ouverture d'un menu "burger" ou avec des conditions appropriées lorsque le site est visité en mobile ou pas.
Deuxième point ; une fonction peut être appelée plusieurs fois pour effectuer le même traitement.
Imaginons que l'on ait besoin d'un chiffre random plusieurs fois dans plusieurs parties du script, on peut mettre le code du random dans une fonction et appeler celle-ci plusieurs fois plutôt que le code lui même :

```
Exemple création :
function randomize(){
return Math.floor(Math.random() * 100);
}
Plusieurs lancement :
var rand1 = randomize();
console.log(rand1) // affiche un chiffre entre 0 et 100, ex : 56
var rand2 = randomize();
console.log(rand2) // affiche un chiffre entre 0 et 100, ex : 30
var rand3 = randomize();
console.log(rand3) // affiche un chiffre entre 0 et 100, ex : 89
var rand4 = randomize();
console.log(rand4) // affiche un chiffre entre 0 et 100, ex : 6
// Cet exemple est simpliste mais imaginez que cette fonction se lance chaque fois que l'utilisateur clique,
on peut cliquer 2000 fois, mais on écrit une seule fois le corps de la fonction, pas 2000.
Dans ce cas, l'attente d'une action et la réutilisation sont la clef.
```

Paramètres (arguments) :
Nous avons vu qu'une fonction peut être considérée comme une boite contenant des instructions, cette fonction peut avoir besoin de paramètres--> de valeurs qui lui seront données au moment de son lancement pour travailler.
Un exemple pour comprendre :
création d'une fonction très basique qui va afficher un nom concaténé à une phrase dans la console lors de son lancement.
Création :

```
function bonjour(){
    console.log("Bonjour Jean bienvenue !");
}
```
Lancement :
bonjour(); // affiche dans la console "Bonjour Jean bienvenue !".
bonjour(); // affiche dans la console "Bonjour Jean bienvenue !".
bonjour(); // affiche dans la console "Bonjour Jean bienvenue !".
Par trois fois, on affiche la même phrase, c'est très bien pour mon ego mais ça ne va pas très loin.

Construisons maintenant la fonction avec un paramètre, celui-ci se met dans les parenthèses :
```
function bonjour(paramNom){
    console.log("Bonjour "+paramNom+" bienvenue !");
}
```
Puis on appelle plusieurs fois en changeant la valeur du paramètre :
bonjour("**Sophie**"); // affiche dans la console "Bonjour **Sophie** bienvenue !".
bonjour("**Fred**"); // affiche dans la console "Bonjour **Fred** bienvenue !"
bonjour("**Choucroute**"); // affiche dans la console "Bonjour **Choucroute** bienvenue !"

Votre fonction est maintenant nettement plus efficace. Grâce à ce simple paramètre, votre fonction adapte automatiquement le message que vous souhaitez délivrer.

Bien entendu un paramètre peut être de n'importe quel type et vous pouvez en mettre plusieurs à la fois.

Exemple :
```
var classHero = ["healer","dps","tank","support"];
function setClassOfPlayer(dataClass,hClass){
    for(var i = 0; i<dataClass.length; i++){
        if(dataClass[i] === hClass){
                bloc de code ;
        }
    }
}
setClassOfPlayer(classHero,"tank");

// on voit bien ici qu'il est possible de mettre plusieurs paramètres dans la fonction.
```
!!Attention il y a un ordre : si vous demandez en paramètre un nombre ou, comme ici, un tableau puis une chaîne et ainsi de suite, lors de la construction, il faut impérativement que les paramètres renseignés lors de l'appel de la fonction soit de même nature.

Exemple :
```
setClassOfPlayer("healer",classHero) ; // renvoie une erreur car c'est un tableau (object) qui est demandé en premier paramètre pas une chaîne(string).
```

Le renvoi de résultat (return) :

Une fonction peut, pour différentes raisons, renvoyer un résultat lorsqu'un traitement s'est terminé correctement ou non, ou un nombre comme dans le cas de la fonction randomize() créée plus haut. Imaginons que vous ayez besoin de déterminer si un champ de formulaire est vide ou plein pour continuer le traitement des données : si le champ est vide alors la fonction renvoie "false" et on stoppe le traitement, sinon on renvoie "true" pour continuer le traitement :

```
Exemple :
var name = "jean" ;
function empty(nameValue){
    if(nameValue !== ""){// on teste si différent de vide (attention un double guillemets sans rien est quand même une chaîne.
            return true ;
    }
    return false ;
}
var test = empty(name); // la valeur de test devient "true" car la fonction a renvoyé son résultat dans la variable;
```

Le mot clef "return" termine la fonction dés qu'il est appelé, le code qui pourrait se trouver ensuite n'est pas exécuté, c'est pour cela que, si la condition répond vrai et que "return true" est envoyé, on ne passe pas par le "return false".

Les fonctions anonymes :

Ce sont des fonctions qui ne seront pas réutilisé plusieurs fois, elle n'ont donc pas besoin de nom, voici la structure de base :

```
function(){
    bloc de code ;
}
```

Ce genre de fonction est beaucoup utilisé pour les "event"(éventements) ou les timer.

```
Exemple :
window.onload = function(){
    un gros bloc de code
}
// En résumé on demande a l'objet window de lancer notre fonction anonyme quand la page sera chargée (c'est enlèvement load), comme la page n 'est chargé une fois à l'ouverture du site on n'a pas besoin de créer un fonction normée qui pourra être rappelé plusieurs fois.
```

Les fonctions auto-exécutées :

Pour diverses raison, notamment le lacement de plugin ou d'API externe comme les scripte d'analyse de google, une gmap ou autre il peut être nécessaire d’utiliser une fonction anonyme qui va s’exécuter automatiquement.

Exemple :

```
var p1 = 10;
var p2 = "test";
(function(param1,param2){
console.log(param1,param2); // la console affichera immédiatement 10 et test
})(p1,p2);
```

première remarque la fonction est contenue dans des parenthèses.
Deuxième remarque les paramètre envoyé a la fonction sont mis dans des parenthèses à la fin de la déclaration.

Retour sur les variables dans une fonction :
Lors du paragraphe sur les variables je vous ai parlé de la portée, petit retour sur la question :
Lors de la création d'une variable dans une fonction, cette variable étant créée dans le contexte de la fonction, en dehors de celle-ci on ne peux pas utiliser la variable.
C'est la même chose pour une autre fonction :
ma fonction fA() contient une variable vA qui ne peut être utilisée dans ma fonction fB() car il n'y a pas de pont entre elles.
Et bien pour les fonctions c'est pareil ! Il est tout à fait possible de créer des fonctions dans des fonctions (en procédural attention à ne pas trop abuser de ce type de construction).
Une fonction comme une variable dépend de son contexte englobant, c'est à dire là ou elle a été crée.

```
Exemple :
function maFunction(){
    function interne(){
            console.log("ici");
    }
    interne(); // remarque la fonction secondaire est lancée dans la fonction parente
}
maFunction(); // affiche "ici".
//
Interne(); // affiche : Uncaught ReferenceError: interne is not defined car la fonction a été lancée hors de
son contexte elle n'est donc pas visible.
```

Pour aller plus loin avec les fonctions :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Fonctions

Les objets

Je vais volontairement peut m'étendre sur les objets car ils font parti de ce que l'on appelle la POO.
Sachez, pour résumer à l'extrême, qu'un objet est une structure de donnée qui possède des propriétés interne (des variable qui lui sont propre) et des methods (des fonction qui lui sont propres) et qu'en javascript enormement d'eslements sont des objets.
C'est pour cette raison que dans ce document j'utilise volontairement le mot de methode pour signifier que celle-ci appartiennent à un objet. Pour exemple "console" est une method de l'objet "window" de même que "alert()".
Dans quelques exemple j'ai aussi utilisé la methode ".pop()" qui permet de supprimer une valeur a la fin d'un tableau, cette methode appartient à l'objet "Array()".

Pour aller plus loin avec les objets :
https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Utiliser_les_objets

Ressources

Sites de références :

https://developer.mozilla.org/fr/docs/Web/JavaScript

https://devdocs.io/javascript/

https://www.guru99.com/interactive-javascript-tutorials.html ,

Forum :

https://stackoverflow.com/

Livres :

Javascript les bons éléments – Douglas Crockford – édition Pearson

Tout Javascript – Olivier Hondermarck – édition Dunod

Apprendre à développer avec Javascript – Christian vigouroux – édition Eni

Apprendre à programmer, algorithmes et conception objet – Christophe Dabancourt – édition Eyrolles