Familiarisation avec

JavaScript

PARTIE **1**

Principes fondamentaux

Introduction

JavaScript est un langage de programmation, utilisé principalement dans le domaine du web pour **dynamiser les pages HTML et y gérer les interactions avec l'utilisateur**.

On le considère comme un langage de script ; il est **géré par le navigateur** (on parle également d'intervention *côté client*) et se greffe en quelque sorte "par-dessus" les pages que celui-ci affiche.

Il a été développé en 1995 (pour être intégré à *Netscape Navigator 2*) puis standardisé par l'ECMA (*European Computer Manufacturers Association*) en 1997 et progressivement intégré à la plupart des navigateurs web ensuite.

Aujourd'hui JavaScript est utilisé sur pratiquement tous ls sites web existant, et connaît une constante évolution. Il existe de nombreuses librairies et *frameworks* visant à en faciliter l'utilisation ; il est devenu accessible à tout intégrateur web sans forcément nécessiter une connaissance trop pointue de son environnement.

Néanmoins quelques principes du fonctionnement de JavaScript méritent d'être abordés, non seulement **pour mieux comprendre la technologie sur laquelle sont basés de nombreux outils que l'on croisera obligatoirement en tant que webdesigner**, mais également comme un premier contact idéal avec la programmation informatique, ses environnements et sa logique.

Mise en place

Un script peut être inclus dans un document HTML au moyen de la balise **<script>**. Cette balise peut contenir des instructions :

```
<script>

  // Instructions JS

</script>
```

ou pointer vers un script externe (fichier .js contenant exclusivement des instructions JavaScript) :

```
<script src = "chemin/vers/script-externe.js" ></script>
```

Même lorsqu'on l'utilise pour pointer vers un script externe et qu'elle est laissée vide, la balise **<script>** nécessite une balise de fermeture **</script>**, contrairement à la balise **<link />** utilisée pour lier des feuilles CSS, qui est *auto-fermante*.

On place généralement ces balises à la fin du corps de document, juste avant la balise fermante **</body>**. Cela permet la prise en compte, par le navigateur, de tous les éléments HTML de la page lors de l'exécution du JavaScript. **Cette prise en compte est nécessaire avant de pouvoir interagir avec le document.**

Un positionnement alternatif est possible **dans l'en-tête** du document HTML, ce qui permettra de charger immédiatement le script.

Mais dans ce cas le HTML n'est lu par le navigateur, puis **affiché à l'utilisateur, qu'après chargement et exécution du JavaScript.** De plus celui-ci risque à nouveau de chercher à interagir avec des éléments encore indéfinis (*undefined*) à ce stade.

Pour contourner ces problèmes on peut utiliser l'attribut **`defer`**, qui permet de **charger un script parallèlement au HTML, tout en attendant que la page soit disponible pour l'exécuter**.

```
<script src = "chemin/vers/script-externe.js" defer ></script>
```

Syntaxe

Lorsqu'on écrit en JavaScript, quelques règles de syntaxe doivent être respectées. Certaines sont obligatoires pour le bon fonctionnement du code, d'autres permettent simplement une meilleure lisibilité, et donc faciliteront la maintenance des scripts.

• Le respect de la casse
JavaScript fait la différence entre majuscules et minuscules.

Prenons par exemple la fonction **`document.getElementById()`** dont on peut remarquer la syntaxe particulière (cette manière d'accoler les mots avec pour chacun une première lettre en capitale est appelée *Camel Case*). Cette fonction ne peut pas s'écrire autrement ; par exemple **`document.getelementbyid()`** ou **`document.GetElementbyid()`** ne sont pas reconnues. Cette règle ne s'applique pas seulement aux fonctions mais à l'ensemble des objets JavaScript.

• Retours à la ligne et points-virgules
Il n'est pas obligatoire, mais tout de même fortement conseillé, de s'en tenir à une seule instruction par ligne. Lorsque plusieurs instructions sont écrites en une seule ligne, un point-virgule est obligatoire pour les séparer.

Bien qu'un retour à la ligne fasse office de point-virgule lorsque le navigateur interprète le code, la convention veut que chaque instruction soit suivie par un point-virgule.

• **Les espaces**

Lors de l'interprétation du code, les espaces ne sont pas pris en compte (sauf ceux faisant partie d'une chaîne de caractères). Leur utilisation aura donc un impact uniquement sur la lisibilité ; un code aéré est préférable, et à partir d'une certaine complexité (imbrication de tests et de boucles, etc) le respect d'une indentation hiérarchisée (retrait horizontal plus ou moins important d'une ligne ou d'un bloc selon son degré d'imbrication) est un atout majeur pour la lisibilité.

• **Les commentaires**

Il peut être très utile d'ajouter à son code des commentaires, c'est à dire des informations qui ne seront pas prises en compte lors de l'interprétation du script ; servant uniquement à décrire ou expliquer un passage du code, pour soi-même ou d'éventuels autres développeurs ayant à travailler plus tard sur le même fichier. La mise en commentaire est également un bon moyen de désactiver temporairement un passage du script sans le supprimer.

Un commentaire sur une seule ligne s'insère en commençant celle-ci par un double slash :

```
// Comme ceci
```

Un commentaire sur plusieurs lignes s'insère de la même manière qu'en CSS, en commençant par **/*** et en se terminant par ***/**.

```
/*
Comme
ceci
*/
```

Debug

Les navigateurs actuels mettent à disposition une console, permettant d'observer en détail le comportement et les propriétés des différents éléments de la page. Cette console s'avère particulièrement utile lorsqu'il s'agit de s'assurer du bon fonctionnement d'instructions JavaScript (ou de comprendre pourquoi, et à quel endroit, elles ne fonctionnent pas).

On y accède généralement, après un clic droit sur la page, via l'option *"Inspecter l'élément"* ou *"Examiner l'élément"*.

On y voit par défaut la structure HTML de la page. Mais à la différence de ce que montre la traditionnelle option *"Afficher le code source"*, la console affichera la structure de la page **en temps réel**, incluant les modifications faites via JavaScript.

Pour une inspection précise des scripts, cliquer sur l'onglet *"Console"* ou *"JavaScript"*. Cela affichera la fameuse console, qui rapportera le cas échéant les erreurs, leur nature, ainsi que le numéro de la ligne à laquelle elles sont détectées.

Il est aussi possible d'inclure dans un script des messages personnalisés, via la fonction **`console.log()`**. Par exemple, les instructions suivantes :

```
var n = 1 ;
console.log("La variable n vaut " + n) ;
```

auront pour effet d'afficher dans la console :

```
La variable n vaut 1
```

Des variantes existent : **`console.debug()`**, **`console.warn()`**, **`console.info()`** et **`console.error()`**. Leur effet sera identique mis à part la présentation visuelle du message dans la console (ajout d'une icône, changement de couleur).

Afficher (sans utiliser dans le script les fonctions évoquées ci-dessus) et manipuler des variables directement dans la console est également possible. En partant du principe que l'on vient d'exécuter le script précédent (initialisation d'une variable n, avec comme valeur 1), et en tapant dans la console :

```
n = n + 1
```

puis

```
console.log("Et maintenant elle vaut " + n)
```

celle-ci affichera :

```
Et maintenant elle vaut 2
```

Données, variables, tableaux

• Les données

Dans un script, il est souvent nécessaire de lire, stocker et manipuler des données. Ces données peuvent consister en :

→ une chaîne de caractères (du texte)
→ un nombre
→ une valeur booléenne, qui sera soit "vraie" (***true***) soit "fausse" (***false***)
→ un élément HTML
→ un objet créé via JavaScript

• Les variables

Pour stocker ces données et y accéder au moment voulu, on utilisera des variables. On déclare une variable au moyen du mot-clé **`var`** suivi du nom que l'on veut lui donner :

```
var maVariable ;
```

On peut ensuite lui assigner une valeur :

```
maVariable = 2 ;
```

et la modifier :

```
maVariable = maVariable + 3 ;
// maVariable vaut maintenant 5
```

Le nom donné à une variable est libre mais certaines règles doivent tout de même être respectées :

→ il ne doit pas contenir d'espaces

→ il ne doit pas contenir de caractères spéciaux (sauf "_" et "$")

→ il ne doit pas contenir de caractères accentués

→ il ne doit pas contenir d'opérateurs mathématiques (**+** , **-** , **/** , *****)

→ il ne doit pas être un mot-clé réservé de Javascript (par exemple **var**)

→ les chiffres sont admis

→ lettres minuscules et majuscules sont admises, et la casse est respectée (par exemple **maVariable** , **MaVariable** et **mavariable** seront trois variables distinctes)

→ de manière générale, il est préférable de donner à une variable un nom logique, reflétant son rôle.

→ Il est possible de déclarer plusieurs variables en une seule ligne :

```
var v1, v2, maVariable, maVariable2 ;
```

→ Il est possible d'assigner une valeur à une variable dès sa déclaration :

```
var v1 = 2 ;
```

→ Les deux principes ci-dessus peuvent être combinés :

```
var v1 = 2 , v2 , maVariable = "bonjour" , maVariable2 ;
```

→ Une variable déclarée mais à laquelle aucune valeur n'a été assignée vaudra par défaut **`null`** (valeur nulle, de type nul). C'est le cas dans l'exemple ci-dessus pour **`v2`** et **`maVariable2`**

→ Désigner une variable qui n'a pas été déclarée retournera la valeur **`undefined`** (indéfini).

• Les tableaux

Lorsqu'au lieu d'une donnée unique on a besoin de manipuler **une liste de données**, il sera plus pratique d'utiliser les tableaux (*arrays*). Le même mot-clé **var** est employé pour déclarer un tableau ; la différence se fait à l'assignation :

```
var mon_tableau = new Array() ;
// Déclare un tableau vide

var mon_tableau2 = new Array( "Valeur 1" , "2" , 3 , "Valeur quatre" , false ) ;
// Déclare un tableau et y assigne 5 valeurs
```

La syntaxe suivante est également admise :

```
var mon_tableau = [] ;
// Double crochet, déclare un tableau vide

var mon_tableau2 = [ "Valeur 1" , "2" , 3 , "Valeur quatre" , false ] ;
// Déclare un tableau et y assigne 5 valeurs
```

Pour désigner une valeur stockée dans un tableau, la syntaxe est la suivante :

```
var entree_N = mon_tableau[N] ;
// N représente l'indice de la valeur recherchée
```

En JavaScript comme dans la plupart des langages de programmation, **le premier indice dans une liste sera 0 et non 1**. Ceci entraîne un décalage qu'il faut bien garder à l'esprit.

Ainsi, dans l'exemple précédent, **`mon_tableau2[1]`** donnera **`"2"`** et non **`"Valeur 1"`**.

On peut obtenir **le nombre d'éléments** contenus dans un tableau en utilisant sa propriété **`length`** :

```
var mon_tableau = [ 1 , 2 , 3 , 4 , "a" , "b" ];

console.log("Mon tableau contient " + mon_tableau.length + "
éléments");
```

• Notes sur les opérations mathématiques

→ Si nous créons deux variables **`var1`** et **`var2`**, en leur assignant des nombres, et que nous voulons ajouter la valeur de **`var2`** à **`var1`**, la syntaxe suivante fonctionnera :

```
var1 = var1 + var2 ;
```

Mais il sera plus fréquent de trouver la syntaxe suivante :

```
var1 += var2 ;
```

Celle-ci s'utilise également avec les autres opérateurs pour soustraire, multiplier ou diviser. On peut bien sûr utiliser des valeurs hors variables :

```
var2 *= 3 ;
// On multiplie var2 par 3
```

→ En programmation, lorsqu'on ajoute 1 à une valeur, on parle d'*incrémentation*. À l'inverse, y soustraire 1 s'appelle une *décrémentation*. Cela s'écrira comme suit :

```
var1++ ;
// On incrémente var1, ce qui équivaut à écrire var1 = var1 + 1

var1--;
// On décrémente var1, ce qui équivaut à écrire var1 = var1 - 1
```

→ L'addition d'une chaîne de caractères et d'un nombre (ou celle de deux chaînes de caractères) s'appelle une ***concaténation***. Le résultat sera une nouvelle chaîne de caractères, dans laquelle on retrouvera bout à bout les valeurs additionnées. Ainsi :

```
"Ma valeur" + 3
```

donnera la chaîne **`"Ma valeur3"`**.

Hormis la concaténation (opérateur **+**), aucune opération mathématique n'est applicable aux chaînes de caractère. On ne peut par exemple pas tronquer une chaîne en y soustrayant une seconde chaîne ni un nombre, et on ne peut pas non plus multiplier ou diviser une chaîne.

```
"Ma valeur" * 3
```

donnera **`NaN`**, qui signifie "*Not a Number*".

Tester une condition

Un script sera souvent amené à n'exécuter certaines instructions que **dans des conditions particulières**. Ces conditions peuvent être testées de la manière suivante :

```
if ( condition ) {

instructions à exécuter si la condition est respectée

}
```

La syntaxe d'une condition se décompose en **trois éléments**, dans l'ordre :

→ Une variable, une expression, un objet, une propriété sur laquelle le test sera effectué

→ Un opérateur de comparaison :

== pour *égal à*
!= pour *différent de*
> pour *supérieur à*
< pour *inférieur à*
>= pour *supérieur ou égal à*
<= pour *inférieur ou égal à*

→ La valeur à laquelle comparer le premier élément.

Exemples :

```
if (var1 >= 3) {
   console.log("Variable supérieure ou égale à 3");
}
if (autre_variable + 2 < 10) {
   console.log("Autre variable logiquement inférieure à 8");
}
```

Il est également possible de préciser des instructions alternatives à exécuter si la condition n'est pas respectée, grâce au mot-clé **else** :

```
if (var2 == false) {
   console.log("var2 est fausse");
} else {
   console.log("var2 est vraie");
}
```

Plusieurs alternatives peuvent être précisées à l'aide de la combinaison **`else if`** :

```
if (var1 >= 3) {

   console.log("Variable supérieure ou égale à 3");

} else if (var1 < 2) {

   console.log("Variable non seulement inférieure à 3 mais aussi inférieure à 2");

} else {

   console.log("Aucune des conditions précédentes n'est respectée. Variable inférieure à 3 et supérieure ou égale à 2");

}
```

Plusieurs conditions peuvent être cumulées grâce aux opérateurs **`&&`** (signifiant *"et"*) et **`||`** (double barre verticale, signifiant *"ou"*) :

```
if (var1 >= 3 && var1 < 4) {
// condition A et condition B doivent être respectées

   console.log("Variable comprise entre 3 (inclus) et 4 (exclus)");

}

if (test_a == true || test_b == true) {
// il suffit qu'une des deux conditions soit respectée

   console.log("La variable test_a est vraie OU la variable test_b est vraie OU les deux sont vraies");

}
```

Il est admis de cumuler plus de deux conditions, ainsi que de les imbriquer à l'aide de parenthèses :

```
if (condition 1 || condition 2 || (condition 3 && condition 4)) {}
```

Ici on doit respecter soit la condition 1, soit la condition 2, soit la réunion des conditions 3 et 4.

```
if (condition 1 && (condition 2 || condition 3) || condition 4) {}
```

Ici on doit respecter soit les conditions 1 et 2, soit les conditions 1 et 3, soit la condition 4.

Pour tester une égalité, l'opérateur à utiliser est **==** (double égal). **Un simple égal n'est pas admis,** car en JavaScript son utilisation est réservée à l'assignation. Ainsi :

```
var v = 3;

console.log(v = 4);
```

Affichera **4**, car l'assignation de la valeur 4 à la variable **v** est tout d'abord exécutée. Alors que :

```
var v = 3;

console.log(v == 4);
```

Affichera **false**, soit le résultat du test de la condition "*v est égal à 4*".

Dans certains cas un opérateur triple peut être utilisé :

=== pour ***strictement égal*** (se vérifie si, à la fois, les deux expressions sont égales et leurs types sont identiques).

!== pour ***strictement différent*** (se vérifie si les deux expressions sont différentes ou si elles sont de deux types différents).

Il n'est pas rare de rencontrer, en guise de condition à tester, les syntaxes suivantes auxquelles il semble à première vue manquer des éléments :

```
if (une_variable) {...}
```

et

```
if (!une_variable) {...}
```

Celles-ci sont en fait des formes raccourcies tout à fait admises ; elles correspondent respectivement à :

```
if (une_variable == true) {...}
```

et

```
if (une_variable != true) {...}
```

Les boucles

Il est possible, à l'aide du mot-clé **`while`**, d'exécuter un bloc d'instructions en boucle **tant qu'une condition est respectée** :

```
var compteur = 0;

while (compteur < 10) {

console.log("Compteur : " + compteur);

compteur++;

}

console.log("Boucle terminée");
```

Ici l'instruction **`console.log("Compteur : " + compteur)`** sera exécutée 10 fois, accompagnée à chaque fois d'une incrémentation (augmentation de 1) de la variable **`compteur`**.

Lorsque celle-ci aura atteint la valeur 10, la condition sera testée une dernière fois et comme elle ne se vérifiera plus, le bloc d'instructions contenu dans la boucle sera ignoré ; le script passera donc à la suite et le message **`"Boucle terminée"`** sera affiché.

Il est important de s'assurer que la condition finit par ne plus être respectée (dans l'exemple précédent, ce rôle est joué par l'incrémentation, dans la boucle, de la variable **`compteur`** qui finira automatiquement par atteindre la valeur 10) **sans quoi la boucle sera exécutée indéfiniment, et le script y restera "bloqué",** sans aucun moyen d'atteindre la suite des instructions.

Il existe une autre manière d'exécuter une boucle, **en fonction de l'évolution d'une variable**. On utilisera le mot-clé **for**, avec la syntaxe suivante :

```
for ( initialisation d'une variable ; condition à tester sur
la variable ; modification de la variable ) {

Instructions à exécuter

}
```

L'exemple précédent adapté à cette syntaxe donnerait :

```
for (var compteur = 0 ; compteur < 10 ; compteur++) {

console.log("Compteur : " + compteur);

}
```

Cette alternative s'avère pratique pour appliquer des instructions à chaque élément d'un tableau :

```
var mon_Tableau = [ "a" , "b" , "c" , "d" ];

for (var i = 0 ; i <= mon_Tableau.length ; i++) {

var e = mon_Tableau[i];

console.log("L'élément " + i + " est " + e);

}
```

Cet exemple affichera :

```
L'élément 0 est a
L'élément 1 est b
L'élément 2 est c
L'élément 3 est d
```

Les fonctions

Lorsqu'une série d'instructions est vouée à être répétée, il devient intéressant d'en faire une fonction. Celle-ci pourra, une fois définie, être appelée autant de fois que nécessaire en économisant du code. Les bénéfices sont multiples : maintenance bien plus facile, script plus rapide à exécuter et plus léger à charger.

• Définir une fonction

Pour définir une fonction, on utilisera le mot-clé **`function`**, suivi du nom que l'on souhaite lui donner, puis de parenthèses contenant les éventuels paramètres requis par la fonction, et enfin, entre accolades, les instructions qu'elle devra exécuter.

```
function ma_fonction(paramètres) {

// instructions à exécuter lors de l'appel de ma_fonction()

}
```

→ Le nom d'une fonction est **soumis aux mêmes contraintes que celui d'une variable**.

→ Une fonction peut être définie comme nécessitant **un, plusieurs ou aucun paramètre**. Ceux-ci sont des variables, réservées à la fonction, que l'on nomme entre les parenthèses pour les appeler ensuite dans le code de la fonction. Lorsque plusieurs paramètres sont listés, ils doivent être séparés par des virgules.

• Appeler une fonction

Pour appeler ensuite cette fonction, il s'agira simplement de saisir son nom, avec les parenthèses, renseignées éventuellement avec les valeurs que l'on veut donner aux paramètres :

```
ma_fonction("abcd" , 1 , false);

mon_autre_fonction();
```

Voici un exemple de définition puis d'utilisation d'une fonction. Dans ce cas elle servira à comparer deux nombres pour afficher dans une boîte de dialogue lequel des deux est le plus grand :

```
// Définition :

function msg_greater(v1 , v2) {

  if (v1 > v2) {

     alert(v1 + " est supérieur à " + v2);

  } else if (v2 > v1) {

     alert(v2 + " est supérieur à " + v1);

  } else {

     alert("Les deux nombres sont égaux");

  }

}

// Appel :

var N1 = 1000 , N2 = 3999 , N3 = N1;

msg_greater(N1 , N2); // affichera : "3999 est supérieur à 1000"

msg_greater(N1 , N3); // affichera "Les deux nombres sont égaux"
```

• Retourner une valeur

Le mot-clé **`return`** permet à une fonction de *retourner* (ou renvoyer) une valeur, c'est à dire qu'**un appel à cette fonction deviendra en quelque sorte cette valeur.** On peut reprendre l'exemple précédent pour le rendre plus flexible :

```
function greater(v1 , v2) {

  if (v1 > v2) {

     return v1;

  } else if (v1 < v2) {

     return v2;

  } else {

     return false;

  }

}

var N1 = 1000 , N2 = 3999;

var G = greater(N1 , N2);
// On assigne à G le résultat retourné par la fonction.

if (G != false) {

  alert("Entre " + N1 + " et " + N2 + ", le plus grand nombre
est " + G);

} else {

  alert("Les deux nombres sont égaux");

}
```

L'intérêt ici est de pouvoir réutiliser la valeur stockée dans **`G`**.

Une fonction peut également elle-même être stockée dans une variable. Il ne faut pas confondre :

```
var v = ma_fonction();
```

et

```
var v = ma_fonction;
```

Dans le premier cas on exécute la fonction puis on en stocke le résultat éventuellement retourné dans la variable **v** (si la fonction ne retourne rien, la variable prendra la valeur **undefined**) et dans le second cas c'est la fonction elle-même qui sera stockée, et sans être appelée.

L'exécution de **return** dans une fonction interrompt celle-ci ; **toute instruction s'y trouvant à la suite est ignorée.** Il est d'ailleurs fréquent de voir l'instruction **return**, sans valeur, utilisée dans une fonction pour en arrêter l'exécution à une certaine étape et sous certaines conditions.

```
function ma_fonction() {

// instructions exécutées dans tous les cas

  if (stop_here == true) {
  // variable définie hors de ma_fonction()

     return;

  }

// instructions exécutées uniquement si stop_here == false

}
```

• La portée des variables

On parle de portée pour indiquer où les variables peuvent être utilisées.

→ Une variable définie hors de toute fonction est utilisable **partout dans le script.** Elle possède une portée *globale*.

→ Une variable définie dans une fonction n'est **accessible que dans celle-ci.** Elle est *locale*.

Ainsi :

```
var v1 = "Cette variable est définie directement dans le script...";

function ma_fonction() {

  v1 += " puis modifiée dans une fonction.";

  var v2 = "Cette autre variable est définie dans une fonction...";

}

v2 += " mais ne peut pas être utilisée hors de cette fonction.";
```

La dernière ligne génèrera une erreur, car **v2** est **undefined**.

→ **Pour créer une variable globale dans une fonction,** il faut la définir sans le mot-clé **var**. Il sera également nécessaire d'exécuter la fonction avant de pouvoir faire référence à cette variable depuis autre part.

```
function ma_fonction() {

  v3 = "Cette variable est définie dans une fonction...";

}

ma_fonction();

v3 += " et à condition que la fonction ait été appelée avant,
elle est utilisable en-dehors.";
```