Partie 1 - Les bases du langage

Table des matières

1. Presentation de Javascript

Historique

Le langage Javascript a été mis au point par Brendan EICH qui travaillait chez Netscape en 1995. En décembre 1995, Sun et Netscape annoncent la sortie de JavaScript.

En mars 1996, Netscape met en œuvre le moteur JavaScript dans son navigateur Web Netscape Navigator 2.0 (l'ancêtre de Firefox).

Netscape soumet alors JavaScript à ECMA International pour la standardisation. Les travaux débutent en novembre 1996, et se terminent en juin 1997 par l'adoption du nouveau standard ECMAScript.

Les spécifications sont rédigées dans le document Standard ECMA-262

Microsoft réagit alors en développant JScript, qu'il inclut ensuite dans Internet Explorer 3.0 en août 1996 … Tous les navigateurs suivent ensuite.

Actuellement, la version du langage supporté par les navigateurs est le ECMAScript 2019. Il y a beaucoup d'évolutions du langage en ce moment : les publications sont mises à jour au moins une fois par an : https://developer.mozilla.org/fr/docs/Web/JavaScript/Language_Resources

Attention : ne confondez pas Java et Javascript !! Dites « JS » si vous voulez raccourci son nom …

Les outils

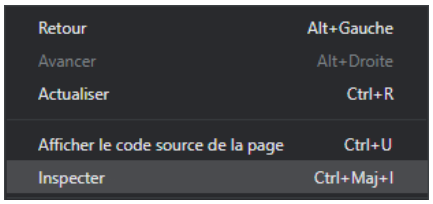
Utilisez un IDE (cela vous aidera) : VS Code, Bracketts, Atom etc.

Utilisez un navigateur avec outils de dev : Chrome ou Firefox

Consultez le site de référence MDN très complet et toujours à jour : https://developer.mozilla.org/fr/docs/Web/JavaScript

La console Javascript

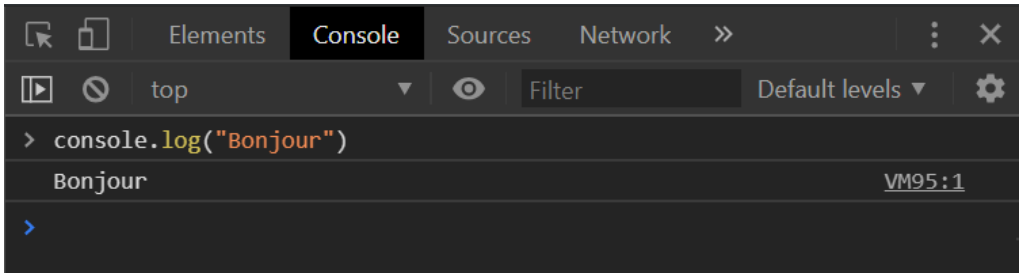
Vous savez faire « Inspecter » par un clic droit dans votre navigateur pour afficher les éléments Html et Css …



Avez-vous remarqué l'onglet « Console » ?

C'est la console Javascript 😊

Vous pouvez interagir directement dans la console, ou, ce qu'on fait très souvent pour débuguer un code javascript, y afficher des messages pour comprendre ce qui se passe dans notre code ou nos variables …



2. Structure du programme

```html
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <link href="styles.css" rel="stylesheet" type="text/css" />
    <title>Javascript</title>
</head>
<body>
    <noscript>Vous devez activer Javascript pour accéder à ce site !</noscript>
    <h1>Mon 1er programme JS</h1>
    <div id="contenu">  </div>

    <script src="script.js"></script> <!--fichier externe Javascript -->
    <script>
        //script JS intégré à la page

        console.log("Coucou !");
    </script>
</body>
</html>
```

Le navigateur charge les éléments référencés dans la page Html dans l'ordre :

1. **Les feuilles de styles CSS** : le CSS est chargé du design du site
2. **Les éléments (contrôles) HTML** : le Html contient la structure du site
3. **Le code Javascript** : le JS, qu'il soit dans un fichier externe ou dans la page Html, doit être placé de préférence à la fin du Body, afin de faciliter le chargement de la page (performance) et de n'autoriser le code qu'une fois les balises Html correctement chargées dans le navigateur.

Remarquez les **commentaires** :

`//` pour commenter une ligne

`/* bloc */` pour commenter tout un bloc de code

On termine les instructions par un **point-virgule**. Ce n'est plus obligatoire depuis ECMAScript 6 (2015) mais ceci reste fortement recommandé !

Le code est interprété **séquentiellement**, une instruction après l'autre.
Le programme s'arrête à la dernière instruction.

3. Fonctions d'interaction avec l'utilisateur

Pour afficher un fragment Html à la suite du flux Html de la page (le DOM) :

```
document.write("<b>Bonjour Nadine !</b>");
```

Pour afficher une alerte (dans une boite de dialogue modale) :

```
alert("Pour afficher un message d'avertissement");
```

Pour poser une question à l'utilisateur (dans une boite de dialogue modale) :

```
let reponse = prompt("Quel est votre prénom ?");
```

4. Variables et valeurs

Une variable est une zone de stockage d'informations dans la mémoire de l'ordinateur. Une variable permet donc de conserver une valeur, une donnée, que l'on veut manipuler dans notre programme. On peut l'imaginer comme une boite à qui on donne un nom et dans laquelle on range des choses.

Il y a 3 manières de déclarer une variable :

```
var prenom; //ancienne déclaration (obsolète)
let prenom = "Nadine"; //déclaration avec définition de la valeur
const pi = 3.14; // constante : ne peut plus être redéfini ensuite
```

Remarque : on évite les caractères accentués et les caractères spéciaux dans les noms de variables !

La convention de nommage des variables est le **camelCase** :

Camel case (de l'anglais, littéralement « casse de chameau ») est une pratique qui consiste à écrire un ensemble de mots en les liant sans espace ni ponctuation, et en mettant en capitale la première lettre de chaque mot.

Les variables peuvent prendre différentes valeurs :

```
let prenom = "Nadine"; // une chaine de caractère

let age = 20; //un nombre (entier)
age = 20.5;   //un nombre aussi (décimal)
age = "20";   //age devient une chaine de caractères !!
// La variable prend le type de son contenu :
// en Javascript, une variable peut changer de type au cours du programme !

let estHumain = true; // un booléen : true ou false

let response;   // reponse est undefined
reponse = null; // null est différent de undefined !
```

Note : Undefined et null permettent de tester si une variable est définie ou non !

Une expression est un morceau de code qui produit une valeur, en combinant des variables, des valeurs et des opérateurs. Lors de l'évaluation d'une expression, les variables sont remplacées par leur valeur.

```
1   const c = 3; // 3 est une expression dont la valeur est 3
2   let d = c; // c est une expression dont la valeur est celle de c (3 ici)
3   d = d + 1; // (d + 1) est une expression dont la valeur est celle de d + 1 (4 ici)
4   console.log(d); // 4
```

Une expression peut comporter des parenthèses qui modifient la priorité des opérations lors de l'évaluation.

```
1   let e = 3 + 2 * 4; // e contient 11 (3 + 8)
2   e = (3 + 2) * 4;   // e contient 20 (5 * 4)
```

Il existe une manière plus concise de faire de l'incrémentation de variable de type nombre :

```
let i = 3;
i += 5; // équivaut à i = i + 5;
i++;    // équivaut à i = i + 1;
i -= 2; // équivaut à i = i - 2;
i--;    // équivaut à i = i - 1;
```

⇨ *Exercices Variables et valeurs …*

5. LES TESTS : IF

Imaginons qu'on souhaite écrire un programme qui fasse saisir un nombre à l'utilisateur, puis qui affiche un message si ce nombre est positif.

```
Saisir un nombre
Si ce nombre est positif
  Afficher un message
```

La syntaxe de cette instruction est la suivante :

```
if(condition) {
    // instructions exécutées quand la condition est vraie
}
```

La paire d'accolades ouvrante et fermante délimite ce que l'on appelle un **bloc de code** associé à l'instruction **if**. Cette instruction représente un **test**.

Ce qui donne le code suivant :

```
const nombre = Number( prompt("Entrez un nombre :") );
if( nombre > 0 ) {
    document.write("Le nombre "+nombre+" est positif");
}
```

Une condition est une expression dont l'évaluation produit une valeur soit vrais soit fausse : c'est une valeur **booléenne**.

```
if (true) {
  // la condition du if est toujours vraie :
  // les instructions de ce bloc seront toujours exécutées
}
if (false) {
  // la condition du if est toujours fausse :
  // les instructions de ce bloc ne seront jamais exécutées
}
```

Les opérateurs de comparaison possibles sont :

Opérateur	Signification
===	Egal à
!==	Différent de
<	Inférieur à
<=	Inférieur ou égal à
>	Supérieur à
>=	Supérieur ou égal à

Dans d'autres langages de programmation, les opérateurs d'égalité et d'inégalité s'écrivent : **==** et **!=**

Javascript supporte également ces opérateurs mais on préfère utiliser les comparaisons strictes du tableau ci-contre …

Attention : la confusion entre **=** et **===** est fréquente ! Vous voilà prévenu 😊

La condition alternative : **ELSE**

```
const nombre = Number( prompt("Entrez un nombre :") );
if( nombre > 0 ) {
    document.write("Le nombre "+nombre+" est positif");
}
else {
    document.write("Le nombre "+nombre+" est nul ou négatif");
}
```

Il est possible d'imbriquer des conditions (autant que l'on veut) :

```
const nombre = Number( prompt("Entrez un nombre :") );
if( nombre > 0 ) {
    document.write("Le nombre "+nombre+" est positif");
}
else if( nombre < 0 ) {
    document.write("Le nombre "+nombre+" est négatif");
}
else {
    document.write("Le nombre est nul");
}
```

6. LES CONDITIONS COMPOSEES : ET, OR, NOT

L'opérateur logique ET : &&

Imaginons qu'on souhaite vérifier qu'un nombre est compris entre 1 et 100. On pourrait imbriquer deux IF, mais on peut aussi utiliser une condition composée de 2 conditions :

```
if((nombre > 0) && (nombre <= 100)) {
    document.write("Le nombre "+nombre+" est compris entre 1 et 100");
}
```

L'opérateur **&&** s'applique à deux valeurs de type booléen. Son résultat est vrai uniquement si les deux valeurs sont vraies :

```
console.log(true && true);   // true
console.log(true && false);  // false
console.log(false && true);  // false
console.log(false && false); // false
```

L'opérateur logique OU : ||

Imaginons maintenant qu'on souhaite vérifier que le nombre est en-dehors de l'intervalle [0,100] :

```
if((nombre < 0) || (nombre > 100)) {
    document.write("Le nombre "+nombre+" est en dehors de l'intervalle [0,100]");
}
```

L'opérateur **&&** s'applique à deux valeurs de type booléen. Son résultat est vrai si au moins une des deux valeurs est vraie :

```
console.log(true || true);   // true
console.log(true || false);  // true
console.log(false || true);  // true
console.log(false || false); // false
```

L'opérateur logique NON : !

Il existe un 3ème opérateur qui permet d'inverser la valeur d'une condition : l'opérateur NON.

```
if( !(nombre > 100) ) {
    document.write("Le nombre "+nombre+" est inférieur ou égal à 100");
}
```

La table de vérité de cet opérateur :

```
console.log(!true);  // false
console.log(!false); // true
```

7. LE CHOIX : SWITCH

Lorsqu'on doit imbriquer plusieurs IF pour une liste de valeurs, une solution plus élégante consiste à effectuer un **switch**. La syntaxe est :

```
switch (expression) {
  case valeur1:
    // instructions exécutées quand expression vaut valeur1
    break;
  case valeur2:
    // instructions exécutées quand expression vaut valeur2
    break;
  ...
  default:
    // instructions exécutées quand aucune des valeurs ne correspond
}
```

Il n'y a pas de limite au nombre de cas possibles.

Le mot-clé **default** placé en fin du switch est optionnel.

Le mot-clé **break**, placé en fin du bloc case, est obligatoire pour sortir du switch.

Par exemple :

```
const meteo = prompt("Quel temps fait-il dehors ?");
switch (meteo) {
  case "soleil":
    console.log("Sortez en t-shirt.");
    break;
  case "vent":
    console.log("Sortez en pull.");
    break;
  case "pluie":
    console.log("Sortez en blouson.");
    break;
  case "neige":
    console.log("Restez au chaud à la maison.");
    break;
  default:
    console.log("Je n'ai pas compris !");
}
```

⇨ *Exercices If et Switch …*

8. LES BOUCLES : FOR ET WHILE

Essayons d'écrire un programme qui affiche tous les nombres entre 1 et 100.

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
...
```

Ça fait beaucoup de lignes … on doit pouvoir faire mieux !!

Pour cela, le langage JavaScript offre la possibilité de répéter l'exécution d'un ensemble d'instructions en plaçant ces instructions à l'intérieur d'une **boucle**.

Le nombre de répétitions peut être connu à l'avance ou dépendre de l'évaluation d'une condition.

A chaque répétition, les instructions contenues dans la boucle sont exécutées. C'est ce qu'on appelle un **tour de boucle** ou encore une **itération**.

La boucle WHILE

La boucle WHILE permet de répéter des instructions **tant qu'une condition est vérifiée**.

```
console.log("Début du programme");
let nombre = 1;
while (nombre <= 5) {
  console.log(nombre);
  nombre++;
}
console.log("Fin du programme");
```

La syntaxe de la boucle WHILE est :

```
while (condition) {
  // instructions exécutées tant que la condition est vérifiée
}
```

A chaque tour de boucle, la condition est évaluée. Si la condition est vraie, le bloc d'instructions est exécuté. Si non, le programme continue juste après le corps de la boucle WHILE (après l'accolade de fermeture)

La boucle FOR

On a fréquemment besoin d'écrire des boucles de type « compteur », c'est-à-dire possédant une variable incrémentée à chaque tour de boucle. Dans ce cas, on utilise la boucle FOR :

```
for (let i = 1; i <= 100; i++) {
  console.log(i);
}
```

Son fonctionnement est un peu plus complexe que la boucle WHILE :

- **L'initialisation** se produit une seule fois au début de l'exécution du FOR : `let i = 1;`
- **La condition** est évaluée avant chaque tour de boucle. Si elle est fausse, le boucle est terminée. Si elle est vraie, on continue. `i <= 100;`
- **L'étape** est réalisée après chaque tour de boucle. `i++`

Par convention, la variable compteur d'une boucle FOR est souvent nommée « i ».

Quand choisir un WHILE ou un FOR ?

Si on peut prévoir à l'avance le nombre de tours de boucles à effectuer, la boucle FOR est le meilleur choix car on connait la condition de fin de la boucle.

Si on ne sait pas combien il y aura de tours, il vaut mieux utiliser le WHILE.

⇨ *Exercices Boucles For et While …*