



# JavaScript

-Niveau débutant & intermédiaire-

## Qu'est-ce-que JavaScript ?

JavaScript est un langage de script **orienté objet**. C'est un langage léger qui doit faire partie d'un environnement hôte (un navigateur web par exemple) pour qu'il puisse être utilisé sur les objets de cet environnement.

JavaScript **côté client** étend ses éléments de base en fournissant des objets pour contrôler le navigateur et le Document Object Model (**DOM**). Par exemple, les extensions du langage côté client permettent de placer des éléments dans un formulaire HTML et de réagir aux événements déclenchés par l'utilisateur (les clics, la saisie d'un formulaire, les actions de navigation...)

JavaScript **côté serveur** étend ses éléments de base avec des objets utiles pour le fonctionnement sur un serveur tels que la possibilité de communiquer avec une base de données, manipuler des fichiers, passer d'une application à une autre...

### **Attention !**

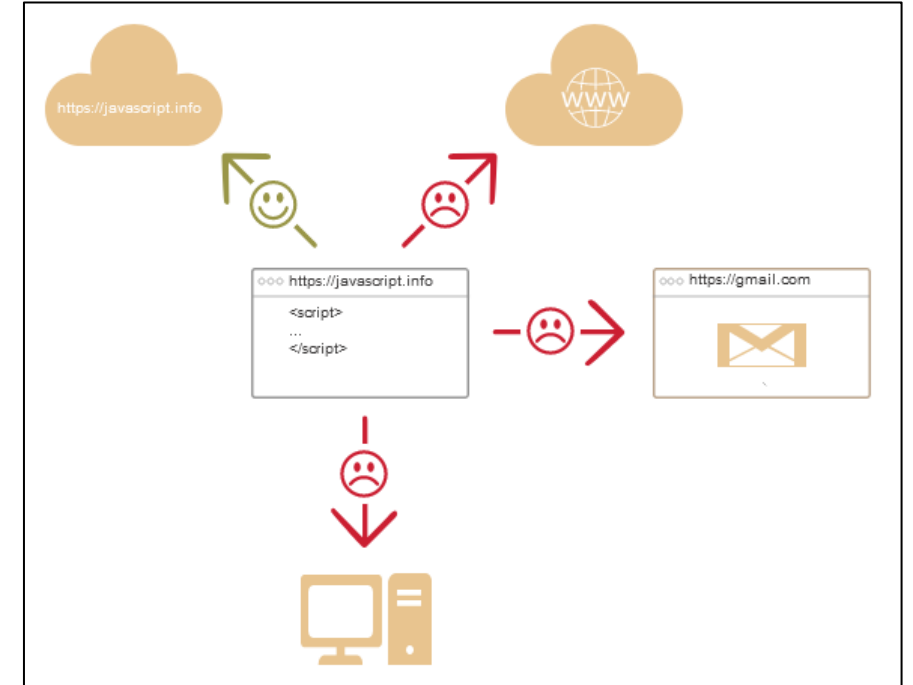
**JavaScript** et **Java** se ressemblent sur certains aspects, mais ils sont fondamentalement **différents** l'un de l'autre.

## Ce que JS peut faire dans le navigateur :

- Ajouter et modifier du code HTML
- Réagir aux actions utilisateurs
- Envoyer des requêtes au serveur
- Gérer les cookies
- Stocker des données client

## Ce que JS ne peut pas faire dans le navigateur :

- Lire et écrire des fichiers sur le disque dur
- Communiquer entre les pages
- Communiquer avec de différents domaines



### **Qu'est-ce-qui rend JS unique ?**

- Intégration complète avec HTML / CSS
- Les choses simples sont faites simplement
- Pris en charge par tous les principaux navigateurs et activé par défaut

### **Conclusion**

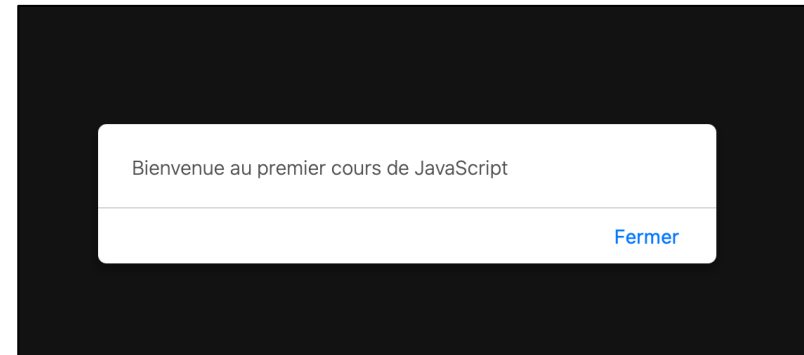
**JavaScript** est :

- Un langage de programmation navigateur, désormais aussi utilisé côté serveur
- Un des langages les plus utilisés dans le web
- La base de beaucoup de framework web

Pour écrire le JavaScript, nous avons **3 possibilités** :

1. Dans l'élément « **head** » du document HTML
2. Dans l'élément « **body** » du document HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>Cours de JavaScript de l'école ri7</title>
  <meta charset="utf-8"/>
</head>
<body>
  <p>1 er cours </p>
  <p>1 er TP/TD</p>
  <p>1 er projet</p>
  <script>
    alert('Bienvenue au premier cours de JavaScript');
  </script>
</body>
</html>
```



1 er cours

1 er TP/TD

1 er projet

3. Dans un **fichier JavaScript séparé**, pour plus de performance, il est recommandé d'utiliser cette méthode.

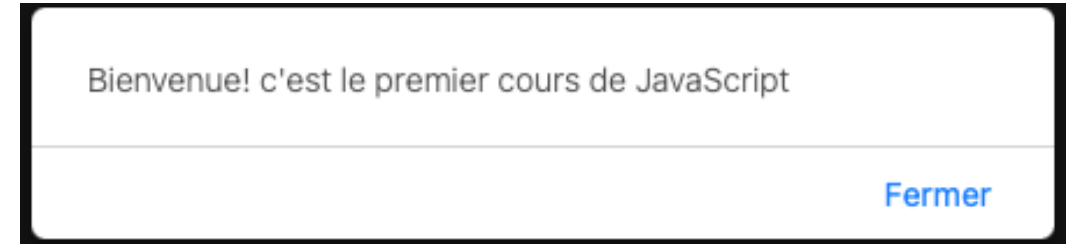
```
cours.html x cours.js x
<!DOCTYPE html>
<html>
<head>
  <title>Cours de JavaScript</title>
  <meta charset="utf-8"/>
  <link rel="stylesheet" href="style.css">
</head>
<body>

  <h1>1 er cours</h1>
  <div>
    <p>1 er TP/TD</p>
    <p>1 er projet</p>

    <script src="cours.js"></script>

  </div>

</body>
</html>
```



```
cours.html x cours.js x style.css
alert("Bienvenue! c'est le premier cours de JavaScript");
```

En JavaScript, les instructions sont appelées **statements** et sont séparées par des points-virgules.

## Les points-virgules :

```
1 alert("Hello");  
2 alert("World");
```

```
1 alert("Hello"); alert("World");
```

## Les erreurs :

```
1 alert("Hello");  
2  
3 [1, 2].forEach(alert);
```

```
1 alert("Hello")  
2  
3 [1, 2].forEach(alert);
```

## Les commentaires :

En JavaScript, il existe 2 types de commentaires: mono-lignes et multi-lignes.

```

<!DOCTYPE html>
<html>
<head>
  <title>Cours de JavaScript</title>
  <meta charset="utf-8"/>
  <link rel="stylesheet" href="style.css">
</head>
<body>

  <h1>1 er cours</h1>
  <div>
    <p>1 er TP/TD</p>
    <p>1 er projet</p>

    <script src="cours.js"> // Ceci est un premier commentaire mon-ligne
/* Ceci est un deuxième commentaire multiligne écrit sur une seule ligne */
/* Ceci est un
troisième
commentaire multiligne
*/
  </script>

  </div>

</body>
</html>

```



## Qu'est-ce-qu'une variable ?

Une **variable** est un **conteneur** qui va servir à stocker temporairement une information. Les **variables** sont utilisées comme des noms symboliques désignant les valeurs utilisées dans l'application.

## Déclaration de variables :

En JavaScript, on déclare une variable avec le mot-clé **var** suivi du nom de la variable. Les noms des variables sont uniques et sont appelés **identifiants**. Ces identifiants doivent respecter certaines règles. Ils doivent contenir que des lettres non accentuées, des chiffres, un **underscore** ( ) ou un symbole **dollar** (\$) et commencer par une **lettre**. Les caractères qui suivent peuvent être des chiffres (0 à 9). Il existe des noms réservés que nous pouvons pas utiliser comme noms de variables (comme « **var** » ).

Il existe **trois** types de déclarations de variables en **JS** :

var

On déclare une variable, éventuellement en initialisant sa valeur.

let

On déclare une variable dont la portée est celle du bloc courant, éventuellement en initialisant sa valeur.

const

On déclare une constante nommée, dont la portée est celle du bloc courant, accessible en lecture seule.

## Assigner une valeur à une variable :

```
1 let message;  
2  
3 message = 'Hello'; // stocke la chaîne de caractères 'Hello' dans la variable
```

```
1 let message;  
2 message = 'Hello!';  
3  
4 alert(message); // affiche le contenu de la variable
```

## Plusieurs manières d'assignation :

```
1 let user = 'John', age = 25, message = 'Hello';
```

```
1 let user = 'John';  
2 let age = 25;  
3 let message = 'Hello';
```

```
1 let user = 'John',  
2   age = 25,  
3   message = 'Hello';
```

## **Règles à respecter....**

Le nom ne doit contenir que des lettres, des chiffres, des symboles « \$ » et « \_ »

Le premier caractère ne doit pas être un chiffre

Attention aux majuscules

Noms réservés 

```
1 let let = 5; // impossible de nommer une variable "let", erreur!  
2 let return = 5; // on ne peut pas la nommer "return" aussi, erreur!
```

Utilisez des noms lisibles

Restez à l'écart des abréviations

N'hésitez pas à créer de nouvelles variables plutôt qu'à réutiliser des variables déjà créées

Les **types de données** permettent en quelque sorte de **classifier** nos différentes **variables**, nous ne pouvons par exemple pas faire des opérations mathématiques avec des caractères, mais plutôt avec des chiffres. Le **Javascript** dispose actuellement de **7 types** de données primitifs: **Number** (les nombres), **BigInt**, **String** (les chaînes de caractères), **Boolean** (les booléens), **Null** (rien), **undefined** (pas défini) et **Symbol**, plus le type **Object** (Objet, peut contenir plusieurs variables de type différents). Nous allons tout de suite voir la différence entre ces type des données.

## 1. Number :

Le type **Number** représente les nombres comme nous les connaissons que ce soit des entiers ou des décimales, en **Javascript**, nous n'avons pas besoin de spécifier si notre nombre est un entier ou décimal, le langage le traite automatiquement. Pour définir donc une variable de type **Number**, on fait:

```
1 let n = 123;  
2 n = 12.345;
```

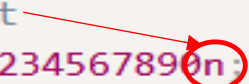
Et nous pouvons aussi faire toutes les **opérations mathématiques**: **+** , **-** , **/** , **\*** , **%**

```
1 alert( "pas un nombre" / 2 ); // NaN, une telle division est erronée
```

## 2. BigInt :

Nombre limité à  $(2^{53}-1)$  (9007199254740991), ou moins que  $-(2^{53}-1)$ . Ce type a été ajouté récemment au langage pour représenter des entiers de longueur arbitraire.

```
1 // le "n" à la fin signifie que c'est un BigInt
2 const bigInt = 1234567890123456789012345678901234567890n;
```



## 3. String :

Les **strings** (chaînes de caractères) sont une suite de caractères, pour créer une chaîne de caractères, il faut toujours la mettre dans des apostrophes. Utilisez des apostrophes simples si votre chaîne contient des apostrophes doubles ou ne contient pas d'apostrophes ('Je suis "contente".') et utilisez des apostrophes doubles si votre chaîne contient des apostrophes simples ("Je suis à l'école"). Nous pouvons aussi écrire ('Je suis à l\'école').

On met un antislash  
avant l'apostrophe

```
1 let str = "Hello";
2 let str2 = 'Single quotes are ok too';
3 let phrase = `can embed another ${str}`;
```

## 3. String :

Les **doubles apostrophes** permettent d'effectuer des fonctionnalités étendues, comme l'intégration de variable ou expression dans la chaîne.

```
1 let name = "John";  
2  
3 // une variable encapsulée  
4 alert( `Hello, ${name}!` ); // Hello, John!  
5  
6 // une expression encapsulée  
7 alert( `the result is ${1 + 2}` ); // le résultat est 3
```

## 4. Boolean :

Les **booléens** sont des variables qui sont soit **vraies** soit **fausses**. Elles ne peuvent donc que prendre ces deux valeurs. On utilise par exemple un booléen pour dire si une **case** à cocher est **cochée** ou non, si un **circuit** est **ouvert** ou fermé...

```
1 let nameFieldChecked = true; // oui, le champ de nom est coché  
2 let ageFieldChecked = false; // non, le champ d'âge n'est pas coché
```

## 4. Boolean :

Les **booléens** sont aussi utilisés pour savoir si **deux variables** sont **égales** ou **différentes**.

```
1 let isGreater = 4 > 1;  
2  
3 alert( isGreater ); // true (le résultat de la comparaison est "oui")
```

## 5. Null :

**Null** est une valeur qui ne représente **rien**. C'est nous même qui l'attribuons à une variable, c'est donc en quelque sorte une absence intentionnelle de valeur pour une variable.

```
1 let age = null;
```

## 6. Undefined :

**Undefined** veut dire qu'une variable a été défini, mais elle n'a aucune valeur. C'est différent de **Null**, quand on déclare une variable, **Javascript** lui affecte automatiquement **undefined** si on **n'associe pas** à cette **variable** une **valeur**.

```
1 let age;  
2  
3 alert(age); // affiche "undefined"
```

Nous avons défini une variable **age** et nous ne lui avons attribué aucune valeur, par défaut la variable **age** est de type **undefined**.

La différence entre **null** et **undefined** est plutôt subtile . Mais il faut savoir que:

- **null** est une valeur que l'on attribue **manuellement**, ça ne se fait pas automatiquement et ça veut dire **ne contient rien**
- **undefined** est un type **automatiquement** attribué à une variable par Javascript, si la **variable** n'est **pas** encore **associée** à une **valeur**.



## 7. Object :

Un **Objet** est une structure contenant des données et des instructions en rapport avec ces données. Un **Objet** correspond parfois à des choses du monde réel, par exemple à une voiture ou à une piste dans un jeu vidéo de course. Les objets servent à stocker des collections de données et des entités plus complexes.

Si la valeur est **null** ou **undefined**, le constructeur **Object** créera et retournera un objet vide, sinon, il retournera un objet du **Type** qui correspond à la **valeur donnée**. Si la valeur est déjà un objet, le constructeur retournera cette valeur.

## 8. L'opérateur « Typeof » :

**Typeof** est un **opérateur** qui retourne une chaîne qui indique le **type d'une variable** (ou d'une expression). Il y a plusieurs syntaxes :

- **opérateur : typeof x**
- **fonction : typeof(x)**

```
var x = 5; // "=" représente un opérateur d'affectation et non pas d'égalité
var nom = "Pascal";
var fort = true;
var n = undefined;
var y = '';
var na = NaN;
var s = null;
var a = 5n;

alert(typeof(x)); //number
alert(typeof(nom)); //string
alert(typeof(fort)); //boolean
alert(typeof(n)); //undefined
alert(typeof(y)); //string
alert(typeof(na)); //number
alert(typeof(s)); //object
alert(typeof(a)); //bigint
```

```
1  typeof undefined // "undefined"
2
3  typeof 0 // "number"
4
5  typeof 10n // "bigint"
6
7  typeof true // "boolean"
8
9  typeof "foo" // "string"
10
11  typeof Symbol("id") // "symbol"
12
13  typeof Math // "object" (1)
14
15  typeof null // "object" (2)
16
17  typeof alert // "function" (3)
```

## **Résumé**

**Number** pour les nombres de toute nature : entier ou réel, les nombres entiers sont limités à  $\pm(2^{53}-1)$ .

**Bigint** pour des nombres entiers de longueur arbitraire.

**String** pour les chaînes de caractères. Une chaîne de caractères peut avoir zéro ou plusieurs caractères.

**Boolean** pour true/false (vrai/faux).

**Null** pour les valeurs inconnues – un type autonome qui a une seule valeur null.

**Undefined** pour les valeurs non attribuées – un type autonome avec une valeur unique undefined.

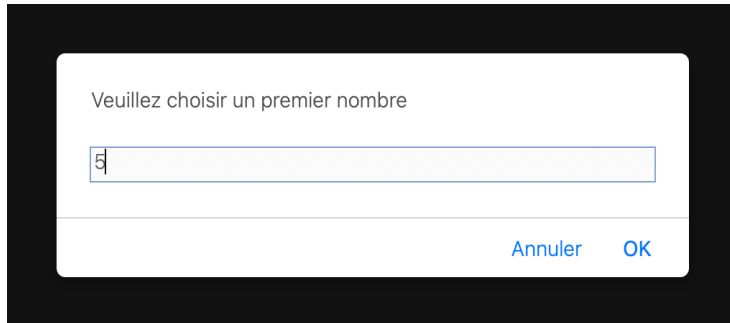
**Object** pour des structures de données plus complexes.

## **Prompt :**

La méthode **prompt()** affiche une boîte de dialogue, éventuellement avec un message, qui invite l'utilisateur à saisir un texte

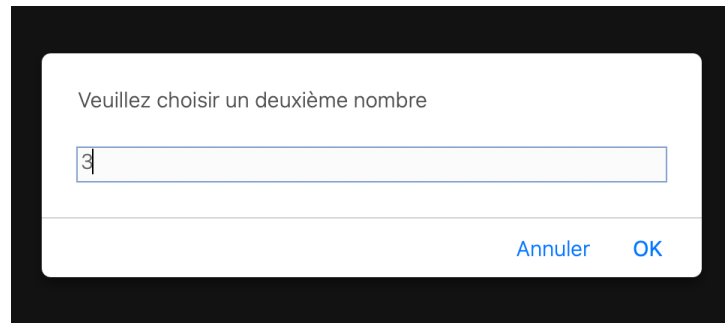
La fonction **prompt()** permet d'interagir avec l'utilisateur, elle lui permet d'entrer du texte dans une boîte de dialogue et pouvoir récupérer ce texte afin de faire des calculs.

```
var x = prompt('Veuillez choisir un premier nombre');  
var y = prompt('Veuillez choisir un deuxième nombre');  
  
var z = x * y;  
  
alert(z);
```



Veuillez choisir un premier nombre

Annuler OK



Veuillez choisir un deuxième nombre

Annuler OK



15

Fermer

Un **opérateur** est un symbole qui produit un résultat en **fonction** de **deux valeurs** (ou **variables**). Le tableau suivant liste certains des opérateurs arithmétiques et logiques les plus utilisés :

Opérateur	Symbole
Comparaison	< , > , <= , >=
Addition	+
Soustraction/multiplication/division	- , * , /
Assignation	=
Égalité	==
Négation	! , !=
ET	&&
OU	

```
1 alert( 2 > 1 ); // true (correct)
2 alert( 2 == 1 ); // false (faux)
3 alert( 2 != 1 ); // true (correct)
```

```
1 alert( 'Z' > 'A' ); // true
2 alert( 'Glow' > 'Glee' ); // true
3 alert( 'Bee' > 'Be' ); // true
```

```
1 alert( '2' > 1 ); // true, la chaîne '2' devient un numéro 2
2 alert( '01' == 1 ); // true, chaîne '01' devient un numéro 1
```

```
1 let result = 5 > 4; // attribue le résultat de la comparaison
2 alert( result ); // true
```

## **Résumé**

- Les opérateurs de comparaison renvoient une valeur logique.
- Les chaînes de caractères sont comparées lettre par lettre dans l'ordre "dictionnaire".
- Lorsque des valeurs de différents types sont comparées, elles sont converties en nombres (à l'exclusion d'un contrôle d'égalité strict).
- Les valeurs null et undefined sont égales == et ne correspondent à aucune autre valeur.
- Soyez prudent lorsque vous utilisez des comparaisons telles que > ou < avec des variables pouvant parfois être null/undefined. Faire une vérification séparée pour null/undefined est une bonne idée.

Les **structures conditionnelles** sont des éléments qui permettent de tester si une expression est vraie ou non et d'exécuter des instructions différentes selon le résultat. La structure conditionnelle utilisée la plus fréquemment est **if ... else**.

```

1 let year = prompt('In which year was ECMAScript-2015 specification published?', '');
2
3 if (year == 2015) alert( 'You are right!' );

```

```

1 let cond = (year == 2015); // l'égalité évalue à vrai ou faux
2
3 if (cond) {
4     ...
5 }

```

La clause « **else** » s'exécute lorsque la condition est fausse.

```

let year = prompt('In which year was the ECMAScript-2015 specification published?', '');

if (year == 2015) {
    alert( 'You guessed it right!' );
} else {
    alert( 'How can you be so wrong?' ); // toute autre valeur que 2015
}

```



## Plusieurs conditions : « else if »

```
if (year < 2015) {  
    alert( 'Too early...' );  
} else if (year > 2015) {  
    alert( 'Too late' );  
} else {  
    alert( 'Exactly!' );  
}
```

## Opérateur ternaire

```
1 let accessAllowed = (age > 18) ? true : false;
```

## Possibilité de chainer les opérations ternaires

```
1 let age = prompt('age?', 18);  
2  
3 let message = (age < 3) ? 'Hi, baby!' :  
4   (age < 18) ? 'Hello!' :  
5   (age < 100) ? 'Greetings!' :  
6   'What an unusual age!';  
7  
8 alert( message );
```

```
1 let accessAllowed;  
2 let age = prompt('How old are you?', '');  
3  
4 if (age > 18) {  
5     accessAllowed = true;  
6 } else {  
7     accessAllowed = false;  
8 }  
9  
10 alert(accessAllowed);
```

## L'opérateur ternaire

Cet opérateur est utilisé comme raccourci pour écrire des conditions. Pour construire une condition ternaire, on va utiliser cet opérateur logique « ? ».

```
var x = 20;  
var y = (x < 10) ? "Vous n'avez pas la moyenne" : "Moyenne";  
  
alert(y); //Moyenne
```

```
var x = 2;  
var y = (x < 10) ? "Vous n'avez pas la moyenne" : "Moyenne";  
  
alert(y); //Vous n'avez pas la moyenne
```



```
if (x < 10) {  
    alert("Vous n'avez pas la moyenne");  
}else{  
    alert("Moyenne");  
}
```

## Switch

L'instruction **switch** va nous permettre d'exécuter un code en fonction de la valeur d'une variable. Elle représente une alternative à l'utilisation d'un **if...else if...else**. Dans un switch, chaque cas va être lié à une valeur précise.

L'instruction **switch** va s'articuler autour de case qui sont des « cas » ou des « issues » possibles. Si la valeur de notre variable est égale à celle du case, alors on exécute le code qui est à l'intérieur.

```
var x = 15;

switch(x){
  case 10 :
    alert("Moyenne");
    break;
  case 18 :
    alert("Très bonne moyenne");
    break;
  case 5 :
    alert("Mauvaise moyenne");
    break;
  default :
    alert("Afficher cette moyenne");
}

//Afficher cette moyenne
```

**Réalisez le TD N°1**

Cette instruction « **while** » permet d'exécuter une instruction tant qu'une condition donnée est vérifiée. Elle s'utilise de la façon suivante :

```
1 while (condition) {  
2   // code  
3   // appelé "loop body" ("corps de boucle")  
4 }
```

```
1 let i = 0;  
2 while (i < 3) { // affiche 0, puis 1, puis 2  
3   alert( i );  
4   i++;  
5 }
```

```
1 let i = 3;  
2 while (i) { // quand i devient 0, la condition devient fausse et la boucle s'arrête  
3   alert( i );  
4   i--;  
5 }
```

Si la condition de sortie n'est pas respectée, on rentre alors dans une boucle infinie!

```
var i = 0;
while (i < 3) {
    alert(i);
    i++; //i=i+1
}
```

/\*

Dans ce cas, on affiche puis on incrémente  
donc on affiche 0 puis on ajoute à chaque fois 1  
jusqu'à 3

i = 0

i=i+1=0+1=1

i=i+1=1+1=2

\*/

```
var i = 0;
while (i < 3) {
    i++; //i=i+1
    alert(i);
}
```

}

/\*

Dans ce cas, on incrémenteon puis on affiche  
donc

i=i+1=0+1=1

i=i+1=1+1=2

i=i+1=2+1=3

\*/

L'instruction « **do ..while** » permet de répéter un ensemble d'instructions jusqu'à ce qu'une condition donnée ne soit plus vérifiée.

« do...while » signifie « faire... tant que ». Elle s'utilise de la façon suivante :

```
1  do {  
2    // corps de la boucle  
3  } while (condition);
```

```
1  let i = 0;  
2  do {  
3    alert( i );  
4    i++;  
5  } while (i < 3);
```

## **Différence entre « while » et « do..while » :**

Dans la boucle **while**, si la condition est fausse dès le début, la boucle ne sera jamais exécutée, en revanche, dans la boucle **do..while**, au moins un passage sera effectué même si la condition est fausse dès le début car dans ce cas la condition va être testée après le passage dans la boucle.

Une boucle **for** répète des instructions jusqu'à ce qu'une condition donnée ne soit plus vérifiée. Elle s'utilise de la façon suivante:

```
1 for (début; condition; étape) {  
2   // ... corps de la boucle ...  
3 }
```

```
1 for (let i = 0; i < 3; i++) { // affiche 0, puis 1, puis 2  
2   alert(i);  
3 }
```

Examinons la déclaration for partie par partie :

partie		
début	let i = 0	Exécute une fois en entrant dans la boucle.
condition	i < 3	Vérifié avant chaque itération de la boucle, en cas d'échec, la boucle s'arrête.
corps	alert(i)	Exécute encore et encore tant que la condition est vraie
étape	i++	Exécute après le corps à chaque itération



## L'instruction break

Lorsque **break** est utilisée, elle provoque la fin de l'instruction **while**, **do-while**, **for**, ou **switch** dans laquelle cette instruction est inscrite (on finit l'instruction la plus imbriquée), le contrôle est ensuite passé à l'instruction suivante.

“boucle infinie + break” est idéale pour les situations où la condition doit être vérifiée.

```
for (i = 0; i < a.length; i++) {  
    if (a[i] === valeurTest) {  
        break;  
    }  
}
```

## L'instruction continue

L'instruction **continue** permet de reprendre une boucle **while**, **do-while**, **for**.

Lorsque **continue** est utilisé, l'itération courante de la boucle (celle la plus imbriquée) est terminée et la boucle passe à l'exécution de la prochaine itération. À la différence de l'instruction **break**, **continue** ne stoppe pas entièrement l'exécution de la boucle. Si elle est utilisée dans une boucle **while**, l'itération reprend au niveau de la condition d'arrêt.

Dans une boucle **for**, l'itération reprend au niveau de l'expression d'incrément pour la boucle.

```
let i = 0;
let n = 0;
while (i < 5) {
  i++;
  if (i === 3) {
    continue;
  }
  n += i;
  console.log(n);
}
// 1, 3, 7, 12
```

Cela n'arrête pas toute la boucle. Au lieu de cela, elle arrête l'itération en cours et force la boucle à en démarrer une nouvelle

## **Résumé**

- 3 types de boucles :

**while** → La condition est vérifiée avant chaque itération.

**do..while** → La condition est vérifiée après chaque itération.

**for** → La condition est vérifiée avant chaque itération, des paramètres supplémentaires sont disponibles

- Pour créer une boucle "infinie", on utilise généralement la construction `while (true)`. Une telle boucle, comme toute autre, peut être stoppée avec la directive **break**.

**Réalisez le TD N°2**

## **Définition :**

Une fonction correspond à un bloc de code nommé et réutilisable et dont le but est d'effectuer une tâche précise. Les fonctions définies dans le langage sont appelées fonctions prédéfinies ou fonctions prêtes à l'emploi car il nous suffit de les appeler pour nous en servir.

## **Objectifs :**

- Éviter de réécrire les mêmes blocs de codes et plutôt créer des fonctions qui vont contenir des séries d'instructions que nous pouvons répéter
- Gain de temps de développement et clarté de lecture
- Facilité de maintenance/développement : une seule modification au lieu de modifier chaque copié-collé de bout de code

```
function addition(a,b){  
    return a + b  
}
```

Par exemple, le JavaScript dispose des fonctions :

Math.random() qui renvoi un nombre aléatoire entre 0 et 1 (1 exclu)

.getElementById() qui renvoi s'il existe un élément de la page web

.replace() qui permet de chercher et remplacer une chaine de caractères par une autre dans un string

## 1. Les variables

Il est indispensable de bien comprendre la notion de « portée » des variables lorsqu'on travaille avec les fonctions en JavaScript. La « portée » d'une variable désigne l'espace du script dans laquelle elle va être accessible. En JavaScript, il n'existe que deux espaces de portée différents :

Global : désigne l'entièreté du script excepté l'intérieur des fonctions

Local : désigne l'intérieur d'une fonction

### **Résumé :**

Une variable globale sera accessible partout même à l'intérieur d'une fonction.

Une variable locale ne sera accessible que dans la fonction où elle est définie.

```
let variableGlobal = 1000;

function multiplication(){
  let variableLocal = 3
  return variableGlobal * variableLocal
}
```

Les différences de portée entre les variables **'var'** et **'let'** en JavaScript :

En effet, lorsqu'on utilise la syntaxe **let** pour définir une variable à l'intérieur d'une fonction en JavaScript, la variable va avoir une portée dite « de bloc » : la variable sera accessible dans le bloc dans lequel elle a été définie et dans les blocs que le bloc contient.

En revanche, en définissant une variable avec le mot clé **var** dans une fonction, la variable aura une portée élargie puisque cette variable sera alors accessible dans tous les blocs de la fonction.

```
let variableGlobal = 1000;
var variableGlobal2 = 500;

//affiche : valeur global, 1000, 500
console.log('valeur global', variableGlobal, variableGlobal2);

function uneFonction(){
  let variableGlobal = 3;
  var variableGlobal2 = 7;
  //Affiche : valeur dans la fonction, 3, 7
  console.log('valeur dans la fonction', variableGlobal, variableGlobal2);
}

//Affiche : valeur global 2 le retour, 1000, 7
console.log('valeur global 2 le retour', variableGlobal, variableGlobal2);
```

## 2. Les valeurs de retour

Une valeur de retour est une valeur renvoyée par une fonction une fois que celle-ci a terminé son exécution.

```
function faisDesMaths(a,b,c){  
  let multiple = a * b;  
  let add = multiple + c;  
  let result = add / 2;  
  //Fais des maths renvoi la valeur calculée  
  return result;  
}  
  
let valeur = faisDesMaths(2,3,4);  
//Affiche : 5  
console.log(valeur)
```



Certaines fonctions prédéfinies vont renvoyer une valeur de retour tandis que d'autres ne vont pas en renvoyer.

Il est utile de savoir si la fonction invoquée renvoie un variable :

- Quoi faire après l'exécution de la fonction
- Recueillir la valeur afin de pouvoir l'utiliser/manipuler

Dans le cas de fonctions personnalisées, nous allons devoir décider si notre fonction va renvoyer une valeur ou pas à l'aide du mot-clé **return**.

À retenir : l'utilisation du mot-clé **return** arrêtera l'exécution de la fonction. Tout bloc de codes après une instruction **return** ne sera pas exécuté.

```
function addition(a,b){
    //renvoi le résultat de a + b
    return a + b
}

let add = addition(5+4)
//Affiche: add 9
console.log('add ', add)

function pasOuLaFonctionDiviser(a,b){
    if(b != 0){
        return a/b
    }else{
        alert('impossible de diviser par zéro')
        return NaN
        console.log('ne s\'affichera jamais et le code est grisé sur VsCode')
    }
    return 'complètement inutile ne sera jamais atteint'
}
```

## 2. Les fonctions anonymes

Des fonctions à qui nous n'avons pas donné de nom.

Utile lors d'un appel unique (un seul endroit) et n'est pas réutilisé.

Exemple : Initialisation d'un programme et chargement de données utiles à son lancement.

Se déclare comme une fonction classique, en omettant son nom.

```
function() {  
    console.log('la fonction anonyme est lancée')  
}
```

Mais comment les appeler ?

- Englober le code de la fonction dans une variable

```
let consol = function() {  
    console.log('la fonction anonyme est lancée depuis une variable')  
}  
consol()
```

- Auto-invoquer la fonction anonyme

```
//fonction anonyme  
(function(){console.log('la fonction anonyme est lancée depuis elle même')}}())  
//fonction nommée  
(function laFunc(){console.log('Auto lancé')}}())
```

Exécuter une fonction anonyme lors du déclenchement d'un événement

```
<button id="leBouton">Clique moi</button>
```

```
let leBouton = document.getElementById('leBouton')
leBouton.addEventListener('click', function(){
  console.log('fonction anonyme lancée depuis un évènement click')
})
```

**getElementById** permet de récupérer un élément de la page web par son id.

**addEventListener** permet d'exécuter un bloc de code lors d'un évènement sur un élément de la page web.

Une fonction qui va s'appeler elle-même au sein de son code.

Comme les boucles elle va permettre d'exécuter une action en boucle jusqu'à ce qu'une condition de sortie soit vérifiée.

```
function decompte(t) {  
  if (t > 0) {  
    console.log(t)  
    return decompte(t - 1)  
  }else{  
    console.log('sortie de la récursion')  
    return t  
  }  
};  
// Affiche : 7, 6, 5, 4, 3, 2, 1, sortie de la récursion  
decompte(7)
```

Les « fonctions fléchées » sont appelées ainsi pour leur syntaxe.

```
let addition = (a,b) => a + b
//Affiche : resultat 7
console.log('resultat', addition(3+4))
```

- Cela revient à écrire:

```
let addition = function(a,b) {
  return a + b
}
```

- Il existe encore des variantes d'écriture pour les fonctions fléchées

- Un seul argument :

```
let puissance = a => a * a
```

Pas besoin de parenthèses

- Sans argument :

```
let donneUneInfo = () => console.log('le feu ça brule')
```

Les parenthèses reviennent mais vides

- Multi-ligne : on note le retour de l'instruction return ainsi que l'utilisation des accolades.

```
let faisDesTrucs = (a, b, c) => { // les accolades permettent
  let premier = a * b           // le multiligne
  let deuxième = premier - c
  //le retour du return
  return deuxième
}
```

Attention! cependant, les fonctions fléchées dans la POO ne permettent pas l'accès à certaines propriétés pratiques.

## Définition

Il existe différentes façons de penser / voir / concevoir son code qu'on appelle « paradigmes »

- **La programmation procédurale** : enchaînement de procédures ou d'étapes qui vont résoudre les problèmes un par un.
- **La programmation fonctionnelle** : qui considère le calcul en tant qu'évaluation de fonctions mathématiques et interdit le changement d'état et la mutation des données.
- **La programmation orientée objet** : concevoir un code autour du concept d'objets.

Un objet est une entité qui peut être vue comme indépendante et qui va contenir un ensemble de variables et de fonctions.

À retenir :

- Il existe différentes façons de concevoir son code qu'on appelle « paradigmes ».
- La plupart des langages supportent aujourd'hui plusieurs paradigmes et le JavaScript supporte chacun des trois principaux paradigmes cités.
- Un paradigme n'est qu'une façon de coder, il est important de comprendre qu'un paradigme n'exclut pas les autres.

## 1. Un objet

```
//La variable hero va contenir un objet
//Elle sera donc considérée elle même comme un objet
let hero = {
  //Ici sont les propriétés de l'objet
  nom: 'Ichbindur',
  age: '9000',
  vie: 150,
  attaque: 80,
  //ici la propriété sac est un tableau
  sac: ['boite vide', 'sandwich moisi'],

  //Ici des méthodes de l'objet
  ditBonjour: function() {
    console.log('Ouais, bonjour')
  },
  porteCoup : function() {
    console.log('porte un coup de ' + this.attaque + ' dégats')
  }
}
//Affiche : Object
console.log(typeof hero)
```



## 1. Un objet

Pour accéder aux propriétés et méthodes de l'objet :

- Utiliser le point

```
//Affiche : Ichbindur  
console.log(hero.nom)  
//Affiche : Ouais, bonjour  
hero.ditBonjour()
```

- Utiliser les crochets [ ]

```
//Affiche : 9000  
console.log(hero['vie'])
```

Pour des propriétés ces deux méthode permettent créer et modifier des valeurs dans l'objet

```
hero.age = 30  
//Affiche : 30  
console.log(hero.age)
```

```
hero["defense"] = 45  
//Affiche : 45  
console.log(hero.defense)
```

## 2. this

```
let hero = {  
  //Ici sont les propriétés de l'objet  
  nom: 'Ichbindur',  
  age: '30',  
  vie: 150,  
  attaque: 80,  
  defense: 45,  
  //ici la propriété sac est un tableau  
  sac: ['boite vide', 'sandwich moisi'],  
  
  //Ici des méthodes de l'objet  
  ditBonjour: function() {  
    console.log('Ouais, bonjour')  
  },  
  porteCoup : function() {  
    console.log('porte un coup de ' + this.attaque + ' dégats')  
  }  
}
```

## 2. this

- Référence à l'objet qui est couramment manipulé
- C'est l'objet lui-même

```
let hero = {  
  //Ici sont les propriétés de l'objet  
  nom: 'Ichbindur',  
  age: '30',  
  vie: 150,  
  attaque: 80,  
  defense: 45,  
  //ici la propriété sac est un tableau  
  sac: ['boite vide', 'sandwich moisi'],  
  
  //Ici des méthodes de l'objet  
  ditBonjour: function() {  
    console.log('Ouais, bonjour')  
  },  
  porteCoup : function() {  
    console.log('porte un coup de ' + this.attaque + ' dégats')  
  },  
  stats: function(){  
    console.log('vie:' + this.vie + 'attaque:' + this.attaque + 'defense:' + this.defense )  
  }  
}  
  
//Affiche : vie: 150 attaque: 80 defense: 45  
console.log(hero.stats)
```

## 3. Constructeur

Une fonction constructeur d'objets est une fonction qui va nous permettre de créer des objets semblables.

Deux étapes s'impose à nous :

- Définir le constructeur d'objet
- Appeler le constructeur à l'aide du mot-clé new

```
function Hero(nom, age, vie, attaque, defense) {  
  //Ici sont les propriétés de l'objet  
  this.nom=nom,  
  this.age= age,  
  this.vie= vie,  
  this.attaque= attaque,  
  this.defense= defense,  
  //ici la propriété sac est un tableau  
  this.sac = [],  
  
  //Ici des méthodes de l'objet  
  this.ditBonjour = function() {  
    console.log('Ouais, bonjour')  
  }  
  this.porteCoup = function() {  
    console.log('porte un coup de ' + this.attaque + ' dégats')  
  }  
}  
  
let nouveauHero = new Hero('ichbindur', 9000, 100, 80, 45)  
//Affiche : Ouais, bonjour  
nouveauHero.ditBonjour()
```

### 3. Constructeur

- L'utilisation du mot clé new crée un objet à partir du constructeur.

`nouveauHero` est donc un objet à part entière.

- L'utilisation du mot clé « this » dans le constructeur permet de remplir les propriété de l'objet avec les arguments passés au constructeur

```
function Hero(nom, age, vie, attaque, defense) {  
  //Ici sont les propriétés de l'objet  
  this.nom=nom,  
  this.age= age,  
  this.vie= vie,  
  this.attaque= attaque,  
  this.defense= defense,  
}
```

### 3. Constructeur

- Nous pouvons donc créer plusieurs objets basés sur le même constructeur.

```
let maurice = new Hero('Maurice', 35, 120, 60, 50)
let odette = new Hero('Odette', 78, 80, 120, 0)
let viktor = new Hero('Viktor', 27, 130, 55, 80)

//Affiche : age de maurice 35
console.log('age de maurice', maurice.age)
//Affiche : dégats de odette 120
console.log('dégats de odette', odette.attaque)
//Affiche : défense de viktor 80
console.log('défense de viktor', viktor.defense)
```

- Se sont 3 instances d'objet du constructeur
- Elles sont indépendantes
- Elles possèdent la même structure, les même méthodes, mais pas les mêmes valeurs

## 4. Les classes

Une classe est un plan général qui va servir à créer des objets similaires

Contient une méthode constructeur qui va être appelée automatiquement dès que nous créons un objet

En POO, tous les objets sont créés en instanciant des classes

Utilisation du mot-clé **class**

```
class Ennemi{
  constructor(vie, attaque){
    this.vie = vie
    this.attaque = attaque
  }
  porteCoup = function () {
    console.log('porte un coup de ' + this.attaque + ' dégats')
  }
}

let premierEnnemi = new Ennemi(150,38)
let deuxiemeEnnemi = new Ennemi(100,60)
//Affiche : porte un coup de 38 dégats
console.log(premierEnnemi.porteCoup())
//Affiche : 100
console.log(deuxiemeEnnemi.vie)
```

## 5. L'héritage

- L'héritage permet de créer une classe enfant qui va hériter des propriétés et des méthodes d'une classe parent.
- L'intérêt majeur est de pouvoir définir de nouvelles propriétés et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.

```
class Ennemi{
  constructor(vie, attaque){
    this.vie = vie
    this.attaque = attaque
  }
  porteCoup = function () {
    console.log('porte un coup de ' + this.attaque + ' dégats')
  }
}

class EnnemiRapide extends Ennemi{
  constructor(vie, attaque, vitesse){
    super(vie, attaque)
    this.vitesse = vitesse
  }
  courir = function(){
    console.log('se rapproche du héros plus vite')
  }
}
```

```
let premierEnnemi = new Ennemi(150,38)
//Affiche : porte un coup de 38 dégats
console.log(premierEnnemi.porteCoup())
let rapidos = new EnnemiRapide(140,40,60)
//Affiche : porte un coup de 40 dégats
console.log(rapidos.porteCoup())
//Affiche : se rapproche du héros plus vite
console.log(rapidos.courir())
```

- L'héritage permet la réutilisation des objets et de leur comportement, tout en apportant de légères variations



## 6. DOM

### Définition

Le DOM (Document Object Model) est une représentation structurée du document sous forme « d'arbre », une arbre d'éléments (balises) créé automatiquement par le navigateur.

Chaque branche de cet arbre se termine un nœud qui va contenir des objets

C'est un standard web

Cela représente l'HTML de la page de la page, mais pas que

Contient un API via l'interface Document

Nous l'avons déjà utiliser lorsque nous cherchons un élément de la page avec : `document.getElementById("idDunElement")`

### Structure

Lors de l'affichage par le navigateur, celui-ci va créer un modèle objet de la page

Ce modèle objet correspond à une autre représentation de la page sous forme d'arborescence dont les terminaisons sont de type `Node`

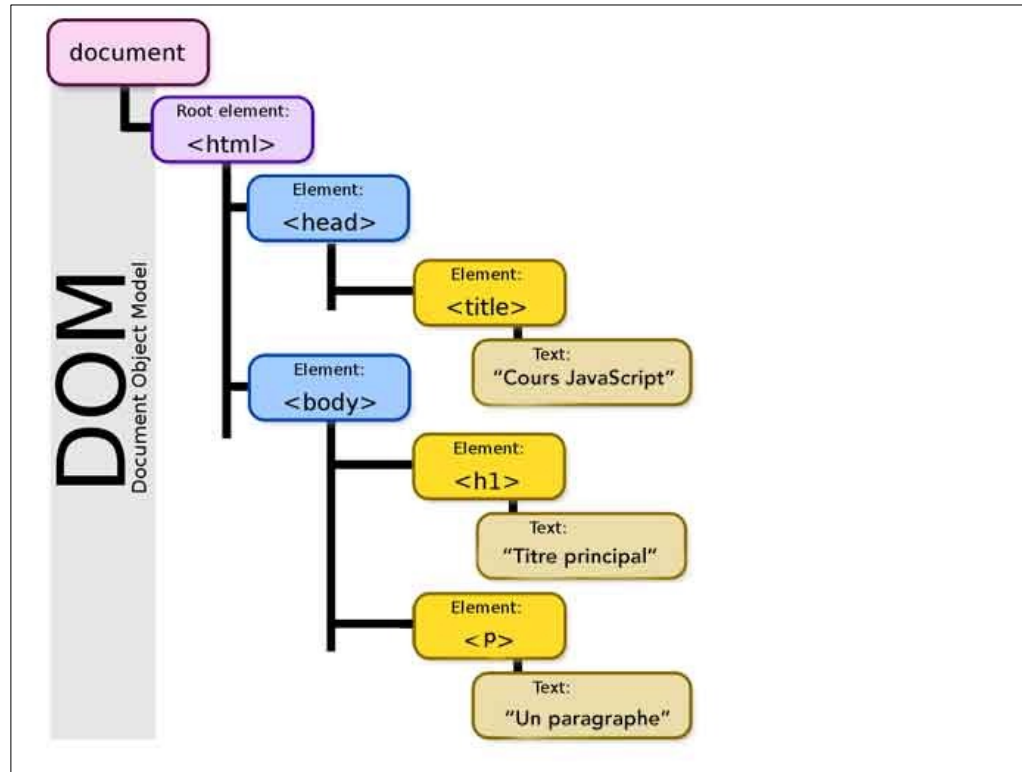
Les navigateurs se base sur cette arborescence notamment pour appliquer les bons styles aux bon éléments

## 6. DOM

### Structure

Voici l'arborescence générée par le navigateur pour afficher la page HTML suivante :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cours JavaScript</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>Titre principale</h1>
    <p>Un paragraphe</p>
  </body>
</html>
```



- "html" représente le noeud racine
- Similaire à l'html de la page source
- Les noeuds "text" sont aussi mentionnés

➤ Chaque entité dans une page HTML va être représentée dans le DOM par un nœud.

## 6. DOM

### Type de nœuds

On différencie les types de nœuds en fonction de si c'est un texte, un élément, un commentaire, etc

Les propriétés et méthodes seront différentes en fonction du type de nœud cible

Voici une liste non exhaustive des nœuds que nous pouvons rencontrer

Constante	Valeur	Description
ELEMENT_NODE	1	Nœud élément (div, p, ...)
TEXT_NODE	3	Nœud de type text
COMMENT_NODE	8	Nœud commentaire
DOCUMENT_NODE	9	Nœud de la page complète
DOCUMENT_TYPE_NODE	10	Nœud du doctype

## 6. DOM

### Type de nœuds

Il existe des types de nœuds dépréciés comme ATTRIBUTE\_NODE (attribut d'un élément)

Il en existe beaucoup d'autres notamment liés au XML

L'un des intérêts majeurs du DOM et des nœuds va être que nous pourrons se déplacer de nœuds en nœuds pour manipuler des éléments en utilisant le JavaScript.

### Les interfaces

Le DOM est un ensemble d'interfaces fonctionnant ensemble

Ces interfaces permettent entre autre d'accéder et manipuler les documents en JavaScript

Il y a une 40aine d'interfaces de bases qui sont composées d'autres interfaces mais nous n'allons pas toutes les voir, c'est juste pour donner une idée sur la complexité de l'arrière plan "navigateur"

## 6. DOM

### Type de nœuds

Il existe des types de nœuds dépréciés comme `ATTRIBUTE_NODE` (attribut d'un élément)

Il en existe beaucoup d'autres notamment liés au XML

L'un des intérêts majeurs du DOM et des nœuds va être que nous pourrons se déplacer de nœuds en nœuds pour manipuler des éléments en utilisant le JavaScript.

### Les interfaces

L'interface `Window`, contenant le document DOM

L'interface `Event` qui représente tout événement qui a lieu dans le DOM

L'interface `EventTarget` qui est une interface que vont implémenter les objets qui peuvent recevoir des événements

L'interface `Node` qui est l'interface de base pour une grande partie des objets du DOM

L'interface `Document` qui représente le document actuel et qui va être l'interface la plus utilisée

L'interface `Element` qui est l'interface de base pour tous les objets d'un document

## **Qu'est ce qu'un évènement en JavaScript ?**

Les événements correspondent à des actions effectuées soit par un utilisateur soit par un navigateur lui-même, par exemple pouvons détecter le clic d'un utilisateur sur un bouton d'un document et afficher une boîte de dialogue ou un texte suite à ce clic. Nous parlerons donc « d'évènement clic »

Il existe de nombreux évènements et nous allons découvrir les évènements les plus courants.

Pour écouter et répondre à un évènement, nous allons définir ce qu'on appelle des **gestionnaires d'évènements**.

## **Les gestionnaires d'évènements :**

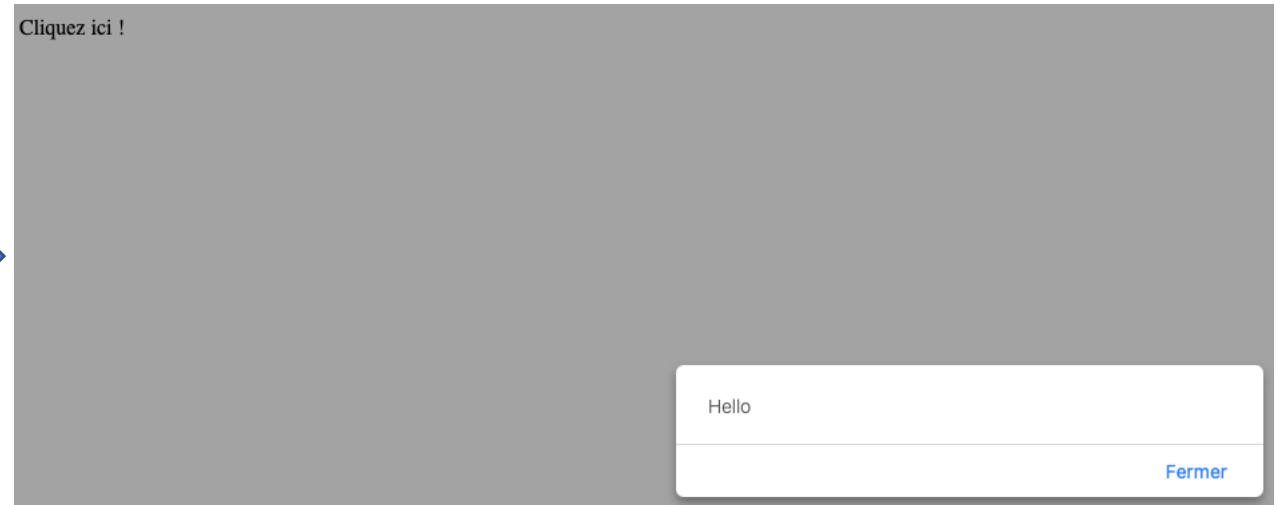
Un gestionnaire d'évènements est toujours divisé en deux parties : une partie qui va servir à écouter le déclenchement de l'évènement, et une partie gestionnaire en soi qui va être le code à exécuter dès que l'évènement se produit.

En JavaScript, il existe différentes façons d'implémenter un gestionnaire d'évènements, les méthodes que nous allons voir dans ce cours sont:

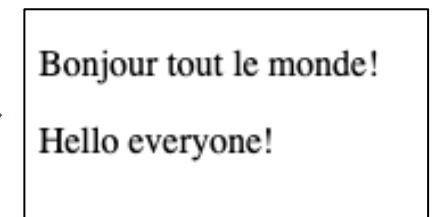
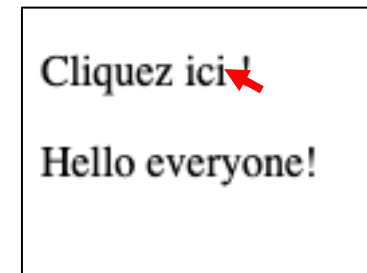
### **1. Les attributs HTML :**

- L'attribut **onclick** pour l'évènement « clic sur un élément » ; se déclenche lorsque l'utilisateur clique sur un élément.

```
<!DOCTYPE html>
<html>
<head>
  <title>Les évènements en JavaScript</title>
  <meta charset="utf-8"/>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <p onclick="alert('Hello');">Cliquez ici !</p>
</body>
</html>
```



```
<!DOCTYPE html>
<html>
<head>
  <title>Les évènements en JavaScript</title>
  <meta charset="utf-8"/>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <p onclick="this.textContent='Bonjour tout le monde!'">Cliquez ici !</p>
  <p>Hello everyone!</p>
</body>
</html>
```

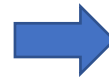


## 1. Les attributs HTML :

- L'attribut **onchange** ; se produit lorsque la valeur d'un élément a été modifiée.

```
<!DOCTYPE html>
<html>
<head>
  <title>Les évènements en JavaScript</title>
  <meta charset="utf-8"/>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <form>
    <p>Sélectionnez une option différente dans cette liste de choix pour déclencher l'évènement
    <b><font color="purple">onchange
    </font></b>.</p>

    <select name=selTest onchange="alert('Index: ' + this.selectedIndex + '\nValeur: ' + this.options[this.selectedIndex].
    value)">
      <option value="Java">Java
      <option value="PHP">PHP
      <option value="JavaScript">JavaScript
    </select>
  </form>
</body>
</html>
```



Sélectionnez une option différente dans cette liste de choix pour déclencher l'évènement **onchange** .

Java

Index: 1  
 Valeur: PHP

Fermer

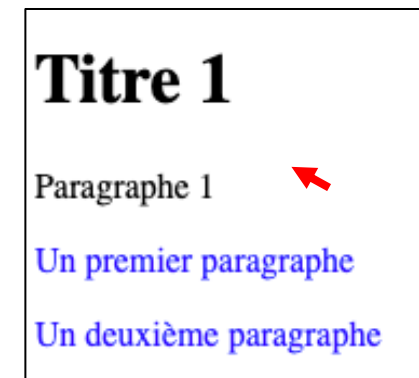
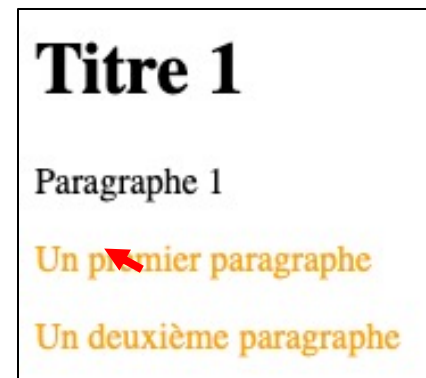


## 1. Les attributs HTML :

- L'attribut **onmouseover** pour l'évènement « passage de la souris sur un élément » ; il se déclenche lorsque l'utilisateur passe le curseur sur un élément.
- L'attribut **onmouseout** pour l'évènement « sortie de la souris d'élément » il se déclenche lorsque l'utilisateur sort son curseur d'un élément.

```
<!DOCTYPE html>
<html>
<head>
  <title>Les évènements en JavaScript</title>
  <meta charset="utf-8"/>
  <link rel="stylesheet" href="style.css">
  <script type="text/javascript"></script>
</head>

<body>
  <h1>Titre 1</h1>
  <p>Paragraphe 1<p>
    <div onmouseover="this.style.color='orange'"
          onmouseout="this.style.color='blue'">
      <p>Un premier paragraphe</p>
      <p>Un deuxième paragraphe</p>
    </div>
</body>
</html>
```



## 2. La méthode **addEventListener** :

- La méthode **addEventListener()** attache une fonction à appeler chaque fois que l'évènement spécifié est envoyé à la cible.

Les cibles courantes sont un Element, le Document lui-même et une Window, mais nous pouvons tout à fait cibler n'importe quel objet prenant en charge les évènements.

**addEventListener()** agit en ajoutant une fonction ou un objet qui implémente EventListener à la liste des gestionnaires d'évènement pour le type d'évènement spécifié sur la cible (EventTarget) à partir de laquelle il est appelé.

```
<!DOCTYPE html>
<html>
<head>
  <title>Les évènements</title>
  <meta charset="utf-8"/>
  <link rel="stylesheet" href="style.css">
</head>
<body>

  <h1 id='gros_titre'>Les évènements</h1>
  <p>Cliquez ici !</p>

  <script>

    var p1 = document.querySelector('p');

    p1.addEventListener('click', changeTexte);

    function changeTexte(){
      this.innerHTML = '<strong>Bravo !</strong>';
      this.style.color = 'orange';
    }

  </script>

</body>
</html>
```



**Titre 1**

**Bravo**

**Titre 1**

**Cliquez ici !**