

Module 133A

Module 133A

Support de cours du module 133A de l'Ecole Professionnelle Technique de la DIVTEC de Porrentruy.

Compétence

Développer, réaliser et tester une application, selon la donnée, avec un langage de scripts côté client.

Objectifs opérationnels

1. Analyser la donnée, développer les fonctionnalités de l'application et s'assurer de la compatibilité des navigateurs.
2. Développer des applications avec les langages HTML, CSS et JavaScript.
3. Choisir et créer des éléments de formulaires adaptés.
4. Assurer l'ergonomie des éléments de formulaires pour toutes les tailles d'écran.
5. Valider les données saisies par l'utilisateur et l'informer des erreurs et corrections à apporter.
6. Commenter la solution de façon compréhensible dans le code source.
7. Appliquer un jeu de test pour valider le bon fonctionnement de la solution.

Environnement de développement

Pour coder en Javascript, un simple éditeur de texte et un navigateur suffisent.

Afin de faciliter le développement d'applications JavaScript, il existe différents outils qui vous permettront de valider la qualité de votre code et d'automatiser certaines tâches. Ci-après une liste de quelques outils.



WebStorm

<https://www.jetbrains.com/webstorm/>

Editeur de code professionnel (gratuit pour étudiants)



Git

<https://git-scm.com/>

Logiciel de gestion de versions



Build software better, together

<https://github.com/>

Service web d'hébergement et de gestion de développement de logiciels



Node.js

<https://nodejs.org/en/>

Serveur JavaScript



ESLint - Pluggable JavaScript linter

<https://eslint.org/>

Analyse la qualité de votre code JavaScript

JavaScript

Les bases

Introduction

JavaScript est un **langage de script, multiplateforme et orienté objet**.

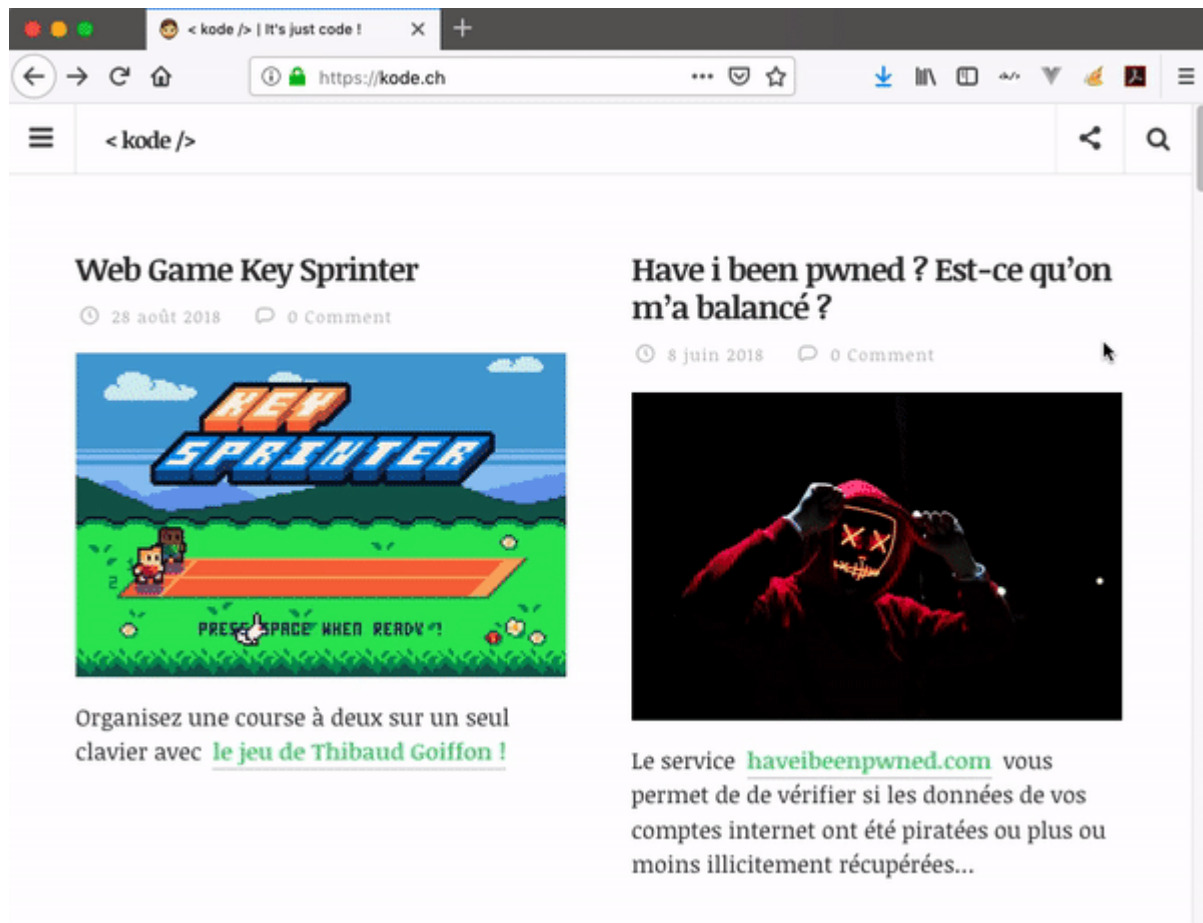
C'est un **langage léger** qui **doit faire partie d'un environnement hôte** (un navigateur web par exemple) pour qu'il puisse être **utilisé sur les objets de cet environnement**.

Quelques généralités sur JavaScript :

- Langage interprété
 - Nécessite un interpréteur (versus. un compilateur)
 - Langage orienté objet
 - Langage à « prototype »
Un prototype est un objet à partir duquel on crée de nouveaux objets
 - Sensible à la casse
 - Confusion fréquente avec Java
 - Aucun lien entre ces 2 langages !
 - Anciennement appelé ECMAScript
 - Standardisé par ECMA (European Computer Manufacturers Association)
-

Ou écrire du JavaScript

Dans la console d'un navigateur



Console JavaScript de Firefox

1. Ouvrir la **console** de votre navigateur `command` + `option` + `J` (Mac) ou `control` + `shift` + `J` (Windows, Linux, Chrome OS) pour ouvrir la **console**.
2. Sélectionner l'**onglet Console**.
3. Écrire l'instruction suivante : `alert("Bonjour les apprentis en JS");`
4. Valider l'instruction avec la touche `↵` Enter

Dans une fichier HTML

Il suffit de placer le code JavaScript dans un élément HTML `<script>` .

Le code JavaScript contenu dans les balises `<script>` est interprété instruction par instruction comme les éléments HTML.

```
1 <h1>Titre de ma page</h1>
2
3 <script>
```

```
4     alert("Bonjour depuis une page HTML !");
5 </script>
6
7 <p>Un petit paragraphe</p>
```



Eviter de mélanger JavaScript et HTML.

Un bon développeur séparera toujours le contenu (HTML), la mise en forme (CSS) et les traitements (JavaScript).

Dans un fichier externe

Généralement on écrit le code JavaScript dans des fichiers portant l'extension `.js`.

Exemple : `panier-achats.js`

Pour intégrer un fichier JavaScript dans un document HTML on utilisera l'élément

`<script>` et l'attribut `src`. Exemple :

```
<script src="panier-achats.js"></script>
```

Où placer la balise `<script>`

On peut placer la balise `<script>` dans l'entête du document `<head>` ou dans le corps `<body>`.

La meilleure pratique consiste à placer ses scripts à la fin du document juste avant la balise de fermeture du corps du document `</body>`.

```
1 <!DOCTYPE html>
2 <html lang="fr">
3   <head>
4     <title>Panier d'achats</title>
5   </head>
```



```
6   <body>
7     <h1>Votre panier d'achats</h1>
8     <p>Cette semaine promotion sur les loutres blanches du Gabon</p>
9
10    <!-- Inclusion des scripts -->
11    <script src="panier-achats.js"></script>
12  </body>
13 </html>
```

Pourquoi à la fin du et pas au début du document, dans l'entête ?

Le navigateur interprète le code de la page et résout les éléments un par un.


Lorsqu'il rencontre un élément `<script>` il va charger tout son contenu avant de passer à l'élément suivant.

L'inclusion des script à la fin du document va donc permettre :

- d'afficher rapidement quelque chose à l'écran. Le navigateur ne doit pas attendre le chargement des scripts avant d'interpréter les autres éléments HTML.
- de manipuler les éléments HTML de la page car tous créés avant l'importation du script.

La directive "use strict"

En ajoutant la directive `"use strict"` au début d'un script, on demande au navigateur de respecter la norme ECMAScript et d'ainsi arrêter le script à la moindre erreur.

 Appliquer "use strict" à tous vos scripts afin d'éviter les auto-corrections des navigateurs. Il vaut mieux stopper un script erroné le plus rapidement possible.

On peut placer la directive au début d'un script ou au début d'une fonction.

Les deux exemples suivant généreront un erreur et le script sera stoppé, car la variable `msg` n'a pas été correctement déclarée.

```
1 <script>
2     "use strict";
3     msg = "Bonjour";
4     alert(msg);
5 </script>
```

```
1 function maFonction() {
2     "use strict";
3     msg = "Bonjour";
4 }
```

Conventions de nommage, JavaScript Style Guide

Beaucoup de guides exposent leur règles de "codage" pour le JavaScript.

Les plus connus sont ceux de AirBnB, GitHub, & Google :

- <https://google.github.io/styleguide/jsguide.html>
- <https://github.com/airbnb/javascript>
- <https://github.com/standard/standard>

Il existe également des outils permettant d'analyser votre code comme **ESLint** & **JSLint**.

A vous de trouvez celui qui vous convient le mieux. Peut importe votre choix, l'important c'est de choisir un style et de le respecter.

Ce support de cours est basé sur les conventions de **Google** et de **JSLint**.

Commentaires & entêtes

```
1 // Ceci est une ligne de commentaires
2
3 /*
4 Ceci est un bloc
5 de commentaires
6 */
```

Entête de document

```
1 /**
2  * @author Steve Fallet <steve.fallet@divtec.ch>
3  * @version 1.6 (Version actuelle)
4  * @since 2018-03-31 (Date de création)
5  */
```

Entête de fonction

```
1 /**
2  * Additionne deux nombres.
3  * @param {number} a - nombre a.
4  * @param {number} b - nombre b.
5  * @return {number} résultat de a + b.
6  */
7 function add(a, b) {
8     return a + b;
9 }
```

Entête de classe

```
1  /** Classe représentant un point. */
2  class Point {
3      /**
4       * Crée un point.
5       * @param {number} x - coordonnée x.
6       * @param {number} y - coordonnée y.
7       */
8      constructor(x, y) {
9          // ...
10     }
11
12     /**
13      * Retourne la coordonnée x
14      * @return {number} La coordonnée x.
15      */
16     getX() {
17         // ...
18     }
19
20     /**
21      * Retourne la coordonnée y
22      * @return {number} La coordonnée y.
23      */
24     getY() {
25         // ...
26     }
27
28     /**
29      * Converti en point une chaine contenant deux nombres séparés par un
30      * @param {string} str - La chaine contenant les deux nombres séparés
31      * @return {Point} Un objet Point.
32      */
33     static fromString(str) {
34         // ...
35     }
36 }
```

Variables et constantes

Conventions

- Nom des variables en **camelCase** : `nomClient`
- Noms des constantes en **MAJUSCULE** avec mots clés séparés par un souligné : `AGE_MAX`
- Une variable doit porter un **nom représentatif de ce qu'elle contient**



JavaScript est **sensible à la casse** : `NomDeFamille` \neq `nomdefamille`

Déclarer des variables et constantes


```
1 // Variables
2 var personneNom = "Dinateur";
3 let personneAge = 22;
4
5 // Constantes
6 const URL = "http://kode.ch";
7 const AGE_MAX = 65;
8 const langues = ['FR', 'EN'];
9 const personne = { age: 20 };
```



Bonne pratique

Ecrire en :

- **majuscule** les constantes contenant des **valeurs** : `const MAX = 33;`
- **minuscule** les constantes contenant des **références** :
`const ids = [12, 44];`

 Les variables JavaScript ne sont **pas typées** !

On peut initialiser une variable avec un entier puis lui affecter une chaîne de caractères sans déclencher d'erreur.

→ **Types de données**

</javascript/introduction/types>

Variables avec **var** ou **let** ?

Il existe deux instructions pour déclarer des variables depuis la sixième édition du standard ECMAScript (ES6 en abrégé).

- **let** permet de déclarer une **variable dont la portée est celle du bloc courant**.
- **var** quant à lui, permet de définir une **variable globale ou locale à une fonction** (sans distinction des blocs utilisés dans la fonction).

```
1 // Avec var
2 function varTest() {
3   var x = 1;
4   if (true) {
5     var x = 2; // c'est la même variable !
6     console.log(x); // 2
7   }
8   console.log(x); // 2
9 }
10
11 // Avec let
12 function letTest() {
13   let x = 1;
14   if (true) {
15     let x = 2; // c'est une variable différente
16     console.log(x); // 2
17   }
18   console.log(x); // 1
19 }
```

Types de données

Les différents types de données

Six types primitifs

- `boolean` pour les booléen : `true` et `false` .
- `null` pour les valeurs nulles (au sens informatique).
- `undefined` pour les valeurs indéfinies.
- `number` pour les nombres entiers ou décimaux. Par exemple : `42` ou `3.14159` .
- `string` pour les chaînes de caractères. Par exemple : `"Coucou"`
- `symbol` pour les symboles, apparus avec ECMAScript 2015 (ES6). Ce type est utilisé pour représenter des données immuables et uniques.

Un type pour les objets `Object`

Les éléments ci-après sont tous de type `Object`

- `Function`
- `Array`
- `Date`
- `RegExp`

Tester le type d'une variable

Les variables peuvent contenir tous types de données à tous moments. Il est donc important de pouvoir tester le type du contenu d'une variable.

L'opérateur `typeof` renvoie une chaîne qui indique le type de son opérande.

```
1 let nombre = 1;
```

```
2 let chaine = 'some Text';
3 let bools = true;
4 let tableau = [];
5 let objet = {};
6 let pasUnNombre = NaN; //NaN (Not A Number) est une valeur utilisée pour r
7 let vide = null;
8 let nonDefini;
9
10 typeof nombre; // 'number'
11 typeof chaine; // 'string'
12 typeof bools; // 'boolean'
13 typeof tableau; // 'object' -- les tableaux sont de type objet.
14 typeof objet; // 'object'
15 typeof pasUnNombre; // 'number' -- Et oui NaN fait partie de l'objet Numbe
16 typeof nonDefini; // 'undefined'
```

Exemple de test de type

```
1 let message = "Bonjour le monde";
2
3 if(typeof message === "string"){
4     alert("c'est une chaine");
5 } else {
6     alert("ce n'est pas une chaine !");
7 }
```

Astuces

Comment savoir si une variable contient un tableau ?

Réponse, on teste si l'objet possède une propriété `length` .

```
1 let tableau = ['je','suis','un','tableau'];
2
3 // Si est un objet et possède un propriété length
4 if(typeof tableau === 'object' && tableau.hasOwnProperty('length')) {
5     alert("C'est un tableau !");
6 } else {
7     alert("Ce n'est PAS un tableau !");
8 }
```

Comment s'assurer qu'une variable est du type number et que c'est un nombre ?

Réponse, utiliser la fonction `isNaN()` qui retourne `true` si la valeur passée en paramètre n'est pas un nombre.

```
1 let age = NaN;
2
3 // Si est de type number et est un nombre valide
4 if(typeof age === 'number' && !isNaN(age)) {
5     alert("C'est un nombre !");
6 } else {
7     alert("Ce n'est PAS un nombre !");
8 }
```

String

Number

Conversions

Convertir des nombres en chaines de caractères

String()

La méthode globale `String()` permet de convertir des nombres en chaines.

```
1 let total = 123.56;
2 String(total); // "123.56"
3 String(123); // "123"
4 String(100 + 23); // "123"
```

.toString()

Autre solution utiliser la méthode `.toString()`.

```
1 let total = 123.56;
2 total.toString(); // "123.56"
3 123.toString(); // "123"
4 (100 + 23).toString(); // "123"
```

Opérateur + concaténation

En utilisant l'opérateur `+` de concaténation, il suffit d'ajouter une chaine au nombre.

```
1 let total = 123.56;
2 total + ""; // "123.56"
3 100 + "123"; // "100123"
4 100 + 23 + ""; // "123"
5 50 + " CHF"; // "50 CHF"
```

Convertir une chaîne de caractères en nombre

Il existe deux méthodes :

- `parseInt(string, base)`
- `parseFloat(string)`

```
1 // Conversion nombre entier en base 10
2 parseInt("35", 10); // 35
3
4 // Conversion en base 2
5 parseInt("01010", 2); // 10
6
7 // Conversion nombre entier (en base 10 si pas de deuxième paramètre)
8 parseInt("22 ans"); // 22
9
10 // Conversion nombre entier
11 parseInt("33.1045"); //33
12
13 // Conversion en nombre flottant
14 parseFloat("33.1045"); //33.1045
15
16 // Conversion en nombre flottant
17 parseFloat("33,1045"); //33 - la virgule n'est pas prise en compte
```

Une bonne pratique pour `parseInt()` est de toujours inclure l'argument qui indique dans quelle base numérique le résultat doit être renvoyé (base 2, base 10...).

Opérateur + unaire

Une autre méthode pour récupérer un nombre à partir d'une chaîne de caractères consiste à utiliser l'opérateur `+`.

```
+"1.1" = 1.1 // fonctionne seulement avec le + unaire
```

Not A Number

TODO - ...

Opérateurs

Les opérateurs numériques en JavaScript sont `+`, `-`, `*`, `/` et `%` (opérateur de reste).

Les valeurs sont affectées à l'aide de `=` et il existe également des opérateurs d'affectation combinés comme `+=` et `-=`.

```
1 // Les deux instructions suivantes sont équivalentes
2 x += 5;
3 x = x + 5;
```

Vous pouvez utiliser `++` et `--` respectivement pour incrémenter et pour décrémenter. Ils peuvent être utilisés comme opérateurs préfixes ou suffixes.



Attention, utiliser `++` et `--` en suffixe ou en préfixe ne fonctionne pas de la même manière.

En **suffixe**, l'opération s'effectuera **après l'affectation** :

```
1 let x = 10;
2 let y = 0;
3
4 y = x++; // y = 10 et x = 11
```

En **préfixe**, l'opération s'effectuera **avant l'affectation** :

```
1 let x = 10;
2 let y = 0;
3
4 y = ++x; // y = 11 et x = 11
```

Opérateur de concaténation de chaînes

L'opérateur `+` permet également de concaténer des chaînes :

```
"coucou" + " monde" // "coucou monde"
```

Si vous additionnez une chaîne à un nombre (ou une autre valeur), tout est d'abord converti en une chaîne. Ceci pourrait vous surprendre :

```
1 "3" + 4 + 5; // "345"  
2 3 + 4 + "5"; // "75"
```

L'ajout d'une chaîne vide à quelque chose est une manière utile de la convertir en une chaîne.

Chaînes de caractères sur plusieurs lignes

Le plus simple est de créer plusieurs chaînes de caractères et de les concaténer.

```
1 let texteYoda = "La peur est le chemin vers le côté obscur : " +  
2   "la peur mène à la colère, " +  
3   "la colère mène à la haine, " +  
4   "la haine mène à la souffrance."  
5  
6 // "La peur est le chemin vers le côté obscur : la peur mène à la colère,
```

Autre solution depuis ES6, utiliser les [Template Literals](#).

Opérateurs de comparaison

Les **comparaisons** en JavaScript se font à l'aide des opérateurs `<` , `>` , `<=` et `>=` . Ceux-ci fonctionnent tant pour les chaînes que pour les nombres.

L'égalité est un peu moins évidente. L'opérateur double égal effectue une équivalence si vous lui donnez des types différents, ce qui donne parfois des résultats intéressants :

```
1 123 == "123"; // true
2 1 == true;    // true
```

Pour éviter les calculs d'équivalences de types, **utilisez l'opérateur triple égal** :

```
1 123 === "123"; //false
2 true === true; // true
```

Les opérateurs `!=` et `!==` existent également.

Conditions

if... else

```
1 // If Else
2 let a = 1;
3 let b = 2;
4
5 if (a < b) {
6   console.log('Juste !');
7 } else {
8   console.log('Faux !');
9 }
10
11
12 // Multi If Else
13 let a = 1;
14 let b = 2;
15 let c = 3;
16
17 if (a > b) {
18   console.log('A plus grand que B');
19 } else if (a > c) {
20   console.log('Mais A est plus grand que C');
21 } else {
22   console.log('A est le plus petit');
23 }
```

L'opérateur (ternaire) conditionnel

L'**opérateur (ternaire) conditionnel** est le seul opérateur JavaScript qui comporte trois opérandes.

Cet opérateur est fréquemment utilisé comme raccourci pour la déclaration `if... else`

```
1 //Initialisation avec condition
2 let solde = 200;
```

```
3 let typeSolde = (solde < 0) ? "Négatif" : "Positif"; //"Positif"
4
5 //Si estMembre est vrai alors retourne 2$ sinon 10$
6 function prixEntree(estMembre) {
7   return (estMembre ? "$2.00" : "$10.00");
8 }
```

Sélections avec `switch`

```
1 let fruit = 'Bananes';
2
3 switch (fruit) {
4   case 'Oranges':
5     console.log('Les oranges sont à 2.55€ le kilo');
6     break;
7   case 'Mangues':
8   case 'Bananes':
9     console.log('Les mangues et bananes sont à 7.70€ le kilo');
10    break;
11  default:
12    console.log('Désolé, nous ne vendons pas de ' + fruit + ' !');
13 }
14
15 // Résultat : 'Les mangues et bananes sont à 7.70€ le kilo'
```

Fonctions

- Bloc de code conçu pour effectuer une tâche particulière.
- Est exécuté quand "quelque chose" l'invoque (l'appelle).
- Seul moyen de créer une nouvelle portée en JavaScript avant ES6
- En JS, les fonctions sont la base de toutes interactions

Créer et appeler une fonction

Si on ne spécifie pas de valeur de retour à une fonction, elle retournera `undefined` .

```
1  function nomFonction() {...}
2
3  function ditBonjour() {
4      console.log("Bonjour !");
5  }
6
7  ditBonjour(); // undefined
8  // Dans la console : "Bonjour !"
```

Paramètres

```
1  function nomFonction(p1,p2,...) {...}
2
3  function ditBonjour(nom, titre) {
4      console.log("Bonjour " + titre + " " + nom + "!");
5  }
6
7  ditBonjour("James", "Monsieur"); // undefined
8  // Dans la console : "Bonjour Monsieur James!"
```

Valeur par défaut des paramètres

Avant ES6, on ne pouvait pas définir de valeur par défaut.

```
1  function ditBonjour(nom, titre) {
2      if (titre === undefined) {
3          titre = "Prince";
4      }
5
6      console.log("Bonjour " + titre + " " + nom + "!");
7  }
8
9  ditBonjour("James"); // undefined
10
11 // Dans la console : "Bonjour Prince James!"
```

Si vous travailler avec ES6, le lien suivant peut vous intéresser :

→ **Paramètres par défaut**

</javascript/javascript-moderne/parametres-par-default>

Retourner une valeur

```
1  function ditBonjour(nom, titre) {
2      if (titre === undefined) {
3          titre = "Prince";
4      }
5
6      return "Bonjour " + titre + " " + nom + "!";
7  }
8
9  ditBonjour("James"); // "Bonjour Prince James!"
10
```

Sortir d'une fonction


En utilisant un `return` on peut forcer la sortie d'une fonction. Tout le code de la fonction situé après le `return` ne sera donc pas exécuté.

```
1 function ditBonjour(nom, titre) {  
2     if (titre === undefined){  
3         return; // Sort, stoppe, la fonction  
4     }  
5  
6     return "Bonjour " + titre + " " + nom + "!";  
7 }  
8  
9 ditBonjour("James"); //undefined
```

Fonctions anonymes

En JavaScript, les fonctions sont des objets, on peut donc stocker des fonctions dans une variable.

```
1 var maFonction = function() {  
2     return "Bonjour de ma fonction";  
3 }  
4 maFonction(); // "Bonjour de ma fonction"
```

 Utiliser un max les fonctions avec des noms, cela vous facilitera la vie lors du débogage.

Si une fonction anonyme déclenche une erreur, dans la console vous aurez comme information :

Erreur déclenchée par "anonymous function"

... oui mais laquelle ???

Fonction “IIFE” Immediately Invoked Function Expression

En plaçant tout votre code dans une "IIFE", cela empêchera vos variables d'entrer en collision avec d'autres scripts. C'est une bonne pratique à respecter si vous utilisez des bibliothèques JS externes.

```
1  (function () {  
2      // Placer tout le code de votre application ici...  
3      console.log("Auto-exécution");  
4  })();
```

Boucles

While

```
1  let i = 0;
2  while (i < 4) {
3    console.log(i);
4    i += 1 // Eviter l'utilisation de "i++". Préférer "++i" ou "i += 1"
5  }
6
7  // 0
8  // 1
9  // 2
10 // 3
```

Do...while

```
1  let i = 0;
2  do {
3    console.log(i);
4    i += 1 // Eviter l'utilisation de "i++". Préférer "++i" ou "i += 1"
5  } while (i < 4)
6
7  // 0
8  // 1
9  // 2
10 // 3
```

For

```
1  //Eviter l'utilisation de "i++". Préférer "++i" ou "i += 1"
2  for (let i = 0; i < 4; ++i) {
```



```
3   console.log(i);
4 }
5
6 // 0
7 // 1
8 // 2
9 // 3
```

For..of

L'instruction `for...of` permet de créer une boucle qui parcourt un objet itérable (Array, Map, Set, String, TypedArray, etc.) et qui permet d'exécuter une ou plusieurs instructions pour la valeur de chaque propriété.

```
1  let animaux = [ ' ', ' ', ' ', ' '];
2
3  for (let animal of animaux) {
4    console.log(animal);
5  }
6
7  // 
8  // 
9  // 
10 // 
```

For...in

L'instruction `for...in` permet d'itérer sur les propriétés d'un objet.

```
1  //Création d'un objet personne
2  let personne = {
3    nom: "Dinateur",
4    prenom: "Laure",
5    age: 33
6  };
```

```
7 //Parcours et affiche le nom et la valeur des propriétés de personne
8 for (let prop in personne) {
9     console.log(prop + " => " + personne[prop]);
10 }
11
12 // nom => Dinateur
13 // prenom => Laure
14 // age => 33
```

Objets

Ajouter, modifier des propriétés

```
1 // Créer un nouvel objet
2 let personne = {};
3
4 // Ajouter un propriété
5 personne.prenom = 'Laure';
6 personne['nom'] = 'Dinateur'; //Autre syntaxe pour l'ajout
7
8 // Accéder à une propriété
9 personne.prenom; // Laure
10 personne.nom; // Dinateur
11
12 // Supprimer une propriété
13 delete personne.nom;
```

Tableaux

Ajouter et retirer des valeurs

```
1 // Crée un tableau vide
2 const monPremierTab = [];
3
4 // Crée un tableau avec valeurs. Peut contenir différents types
5 const monTab = [monPremierTab, 33, true, 'une chaine'];
6
7 // Retourne un élément spécifique du tableau
8 monTab[1]; // Retourne 33
9
10 // Changer une valeur
11 monTab[1] = "ok";
12
13 // Ajouter une valeur à la fin d'un tableau
14 monTab[monTab.length] = 'nouvelle valeur';
15
16 // Ajouter une ou plusieurs valeurs à la fin d'un tableau
17 monTab.push('fromage', 'pain');
18
19 //Ajouter une ou plusieurs valeur au début du tableau
20 monTab.unshift('poivre', 'sel');
21
22 // Récupérer et supprimer le dernier élément d'un tableau
23 let dernier = monTab.pop();
24
25 // Récupérer et supprimer le premier élément du tableau
26 let premier = monTab.shift();
27
28 // Récupérer et supprimer un sous tableau
29 // Premier paramètre position de départ, 2e paramètre le nombre d'éléments
30 monTab.splice(3, 2); //Reourne et supprime le 4e et 5e élément
```

Parcourir un tableau

instruction for

```

1  const animaux = [ ' ', ' ', ' ', ' '];
2
3  // Boucle for classique (éviter i++ et utiliser ++i ou i+=1)
4  for (let i = 0; i < animaux.length; ++i) {
5      console.log(animaux[i]);
6  }
7
8  // 
9  // 
10 // 
11 // 

```

Avec index

```

1  const animaux = [ ' ', ' ', ' ', ' '];
2
3  for (const [index, animal] of animaux.entries()) {
4      console.log(index, animal);
5  }
6
7  // 0 
8  // 1 
9  // 2 
10 // 3 

```

instruction for...of

```

1  const animaux = [" ", " ", " ", " "];
2
3  // Itération avec for..of
4  for (let animal of animaux) {
5      console.log(animal);
6  }
7
8  // 
9  // 
10 // 
11 // 

```

Méthode `forEach()`

```
1  const animaux = [" ", " ", " ", " "];
2
3  // Méthode forEach avec fonction anonyme (depuis ES5 seulement)
4  animaux.forEach(function(animal) {
5      console.log(animal);
6  });
7
8  // 
9  // 
10 // 
11 // 
```

Exemples



133A - Parcourir un tableau

<https://codepen.io/fallinov/pen/BqEJgp?editors=0011>

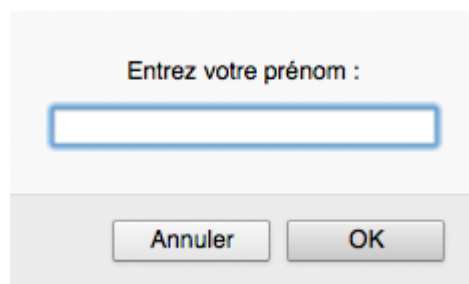
Interactions avec l'utilisateur

Saisie et affichage à l'écran

Maintenant que nous savons utiliser des variables, nous pouvons écrire des programmes qui échangent des informations avec l'utilisateur.

```
1 const prenom = prompt("Entrez votre prénom :");  
2 alert(`Bonjour, ${prenom}`);
```

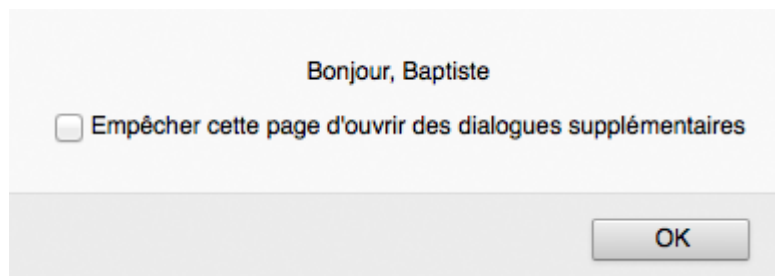
A l'exécution, une première boîte de dialogue apparaît pour demander la saisie du prénom.



Cette boîte est le résultat de l'exécution de l'instruction JavaScript

```
prompt("Entrez votre prénom :") .
```

Après saisie du prénom, une seconde boîte affiche un "bonjour" personnalisé.



La valeur saisie dans la première boîte de dialogue a été stockée dans une variable de type chaîne nommée `prenom`. Ensuite, l'instruction JavaScript `alert()` a déclenché l'affichage de la seconde boîte, contenant le message d'accueil.

Afficher un message à l'utilisateur

Nous avons vu dans les précédents chapitres que l'instruction JavaScript `console.log()` permettait d'afficher une information.

On peut donc utiliser soit `console.log()`, soit `alert()` pour afficher des informations à l'utilisateur. Contrairement à `alert()`, `console.log()` ne bloque pas l'exécution du programme, ce qui en fait souvent un meilleur choix.

Il est possible d'utiliser `console.log()` pour afficher plusieurs valeurs simultanément, en les séparant par des virgules.

```
1  const temp1 = 36.9;
2  const temp2 = 37.6;
3  const temp3 = 37.1;
4  console.log(temp1, temp2, temp3); // Affiche "36.9 37.6 37.1"
```

Saisie de nombres

Quel que soit le texte saisi, l'instruction `prompt()` **renvoie toujours une valeur de type chaîne**. Il faudra penser à convertir cette valeur avec l'instruction `Number()`, `parseInt()` ou `parseFloat()`, si vous souhaitez ensuite la comparer à d'autres nombres ou l'utiliser dans des expressions mathématiques.

```
1  // saisie est de type chaîne
2  const saisie = prompt("Entrez un nombre : ");
3  // transforme saisie en nombre et l'affecte à nb
4  const nb = Number(saisie);
```

Il est possible de combiner les deux opérations (saisie et conversion) en une seule ligne de code, pour un résultat identique :

```
const nb = Number(prompt("Entrez un nombre : "));
```

Ici, le résultat de la saisie utilisateur est directement converti en une valeur de type nombre par l'instruction `Number()` et affecté à la variable `nb` .

Timers & Intervalles

setTimeout()

`setTimeout(function, delai)` permet de définir un « minuteur » (*timer*) qui **exécute une fonction** ou un code donné **après la fin du délai indiqué en millisecondes**.

```
1  function bonjour() {
2      alert('Bonjour !');
3  }
4
5  // Créer un timer et stocke son ID dans timerBonjour
6  // Le timer attendra 5000 millisecondes avant d'appeler la fonction bonjour
7  let timerBonjour = window.setTimeout(bonjour, 5000);
8
9  // Annule le timer correspondant à l'ID passé en paramètre
10 window.clearTimeout(timerBonjour);
```

Exemple

```
1  <button onclick="startBonjour();">
2      Affiche une alerte après 3 secondes...
3  </button>
4  <button onclick="stopBonjour();">
5      Annuler l'affichage
6  </button>
7
8  <script>
9      let timerBonjour;
10
11     function bonjour() {
12         alert("Bonjour !");
13     }
14
15     // Crée un timer qui appelle bonjour() après 3 secondes
16     // Stocke l'ID du timer dans la variable timerBonjour
17     function startBonjour() {
18         timerBonjour = window.setTimeout(bonjour, 3000);
19     }
20
21     // Annule le timer timerBonjour
```

```
22 function stopBonjour() {
23   window.clearTimeout(timerBonjour);
24 }
25 </script>
```



133-JS-SetInterval

<https://codepen.io/fallinov/pen/MzEdJV?editors=0010>

setInterval()

`setInterval(function, delai)` appelle une fonction **de manière répétée**, avec un certain **délai** fixé entre chaque appel.

```
1 function bonjour() {
2   alert('Bonjour !');
3 }
4
5 // Créer un intervalle et stocke son ID dans intervalleBonjour
6 // L'intervalle appellera bonjour() toutes les 5000 millisecondes
7 let intervalleBonjour = window.setInterval(bonjour, 5000);
8
9 // Annule l'intervalle correspondant à l'ID passé en paramètre
10 window.clearInterval(intervalleBonjour);
```

Exemple

```
1 <div>
2   <p>pin-pon, pin-pon, pin-pon ...</p>
3 </div>
4
5 <button onclick="changeCouleur();">Start</button>
6 <button onclick="stopChangeCouleur();">Stop</button>
7
8 <script>
9   let intervalleCouleur;
10
```

```
11 // Crée un intervalle qui appelle flashText() toute les 500 millisecondes
12 function changeCouleur() {
13     intervalleCouleur = setInterval(flashText, 500);
14 }
15
16 function flashText() {
17     // Récupère 1er paragraphe du document
18     let para = document.querySelector("p");
19
20     // Change la couleur du texte en rouge ou en bleu
21     if (para.style.color === "red") {
22         para.style.color = "blue";
23     } else {
24         para.style.color = "red";
25     }
26 }
27
28 // Annule l'intervalle
29 function stopChangeCouleur() {
30     clearInterval(intervalleCouleur);
31 }
32 </script>
```



133-JS-SetInterval

<https://codepen.io/fallinov/pen/MzEdJV?editors=0011>

Manipuler une page Web via le DOM

Introduction

Dis papa c'est quoi le DOM ?

Le DOM (Document Object Model) est un objet JavaScript représentant le document HTML (ou XML) actuellement chargé dans le navigateur.

Dans cet objet, le document `Document` y est **représenté comme un arbre nodal**, chaque nœud `Node` représentant une partie du document.

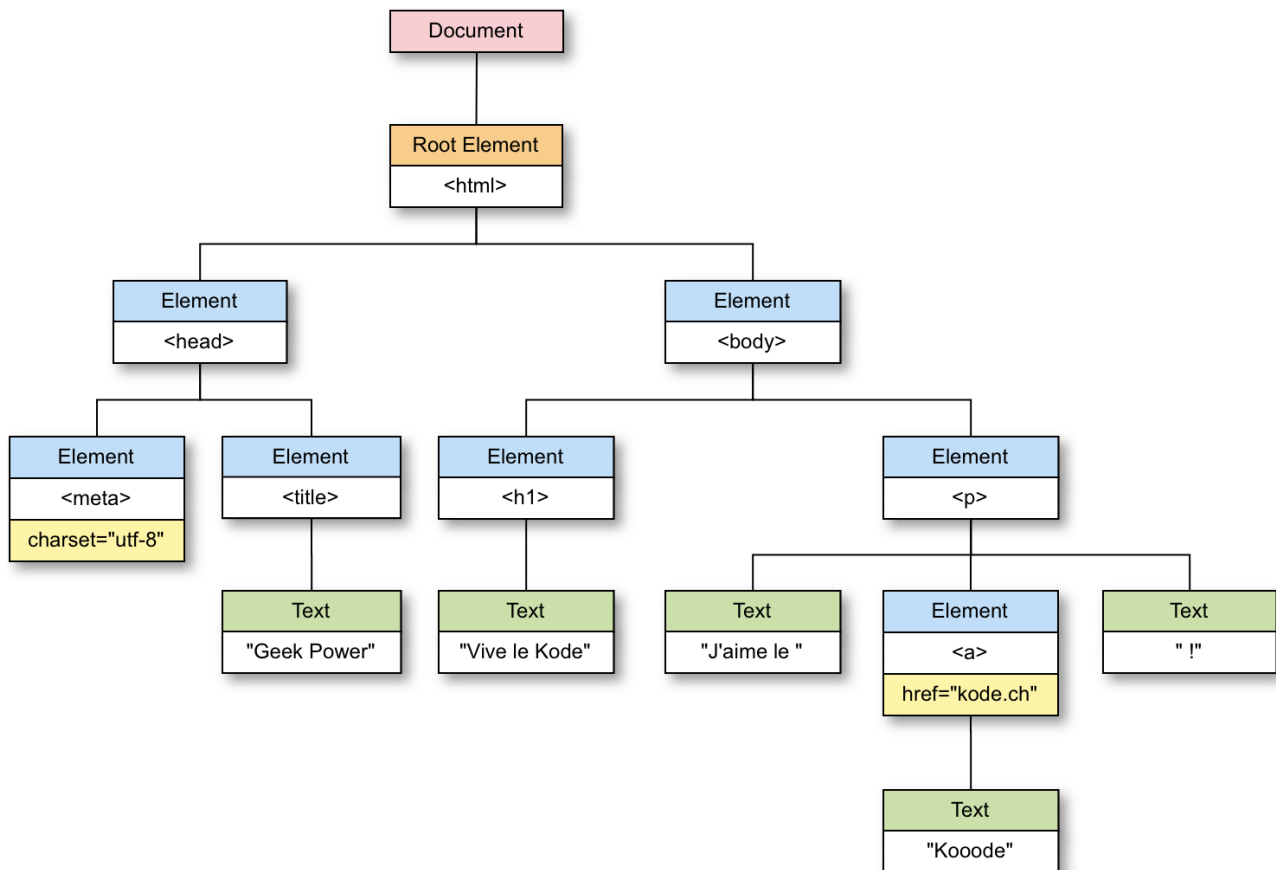
Il existe trois principaux type de nœud :

- `Element` : un élément HTML `<p>`, `<h1>`, `<body>`, ``, ...
- `Text` : chaîne de caractères `"C'est pas faux !"`
- `Comment` : commentaire HTML `<!-- Je suis un simple commentaire -->`

Exemple

Ci-après le code source d'un document HTML et sa représentation sous forme d'arbre nodal de type DOM.

```
1  <!doctype html>
2  <html>
3    <head>
4      <meta charset="utf-8">
5      <title>Geek Power</title>
6    </head>
7
8    <body>
9      <h1>Vive le Kode</h1>
10     <p>J'aime le <a href="http://www.kode.ch">Kooode</a> !</p>
11   </body>
12 </html>
```



Le DOM sous forme d'un arbre nodal du DOM

Et JavaScript dant tou ça ?

Grâce au DOM, JavaScript à le pouvoir de :

- **Récupérer un élément** HTML du document `<h1>`, `<p>`, `<a>`, ...
- **Naviguer entre les éléments** en récupérants ses éléments fils, parents ou voisins (frères)
- **Modifier un élément** en changeant :
 - son **contenu texte** `"texte"` **ou HTML** `"Yoda"`
 - son **style CSS** `fontSize`, `backgroundColor`, `border`, ...
 - ses **attributs** `href`, `class`, `src`, ...
 - ses **événements** `click`, `submit`, `mouseover`, `load`, ...
- **Créer un élément** et l'ajouter au document
- **Supprimer un élément** HTML du document

Ces manipulations sont présentées dans les chapitres suivants.

Accéder aux éléments

On peut rechercher, accéder, aux éléments du document de deux manières :

1. En recherchant **dans tous le document**, en utilisant l'objet `document` .
2. En recherchant **depuis un noeud spécifique** de type `Element` .

La deuxième méthode est plus efficace, puisqu'elle ne nécessite pas un parcours complet du document.

 L'objet `document` représente l'élément `<html>` de la page.

`document.getElementById()`

Ne peut être appelée qu'avec l'objet `document` .

Renvoie **un objet** `Element` représentant l'élément dont l' `id` correspond à la chaîne de caractères passée en paramètre.

```
1 // Renvoie l'élément avec l'id "menu" <nav id="menu">...</nav>
2 const menu = document.getElementById("menu");
```

`Element.getElementsByClassName()`

Peut être appelée avec l'objet `document` ou un objet de type `Element` .

Retourne un **tableau** (`HTMLCollection`) contenant une référence sur tous les éléments ayant les **noms de classes** passés en paramètre.

```
1 // Renvoie un tableau de tous les éléments du document
2 // appartenant à la classe rouge
3 const elementsRouges = document.getElementsByClassName("rouge");
4
5 // Renvoie un tableau de tous les enfants de l'élément spécifié
6 // appartenant aux classes rouge ET gras
7 const elementsRougesGras = monElement.getElementsByClassName("rouge gras")
```



133A - getElementsByClassName

<https://codepen.io/fallinov/pen/BqPNjK>

Exemple de récupération et parcours d'éléments

Element.getElementsByClassName()

Peut être appelée avec l'objet `document` ou un objet de type `Element` .

Retourne un **tableau** (HTMLCollection) contenant une référence sur tous les éléments portant le **nom de balise** donné passé en paramètre.

```
1 // Renvoie un tableau de tous les éléments <li> du document
2 const elementsDeListes = document.getElementsByTagName("li");
3
4 // Renvoie un tableau des éléments <strong> enfants de monElement
5 const taches = monElement.getElementsByTagName("strong");
```

Element.querySelector()

Peut être appelée avec l'objet `document` ou un objet de type `Element` .

Retourne le **premier** `Element` dans le document correspondant au **sélecteur CSS** - ou groupe de sélecteurs - spécifié(s), ou null si aucune correspondance n'est trouvée.

```
1 // Revoie le premier paragraphe du document
2 const premierPara = document.querySelector("p");
3
4 // Renvoie le premier élément du document correspondant à l'un des sélecteurs
5 // 'img.rouge, img-jaune' (images appartenant à la classe rouge OU jaune)
6 const imgRougeOuJaune = document.querySelector("img.rouge, img.jaune");
7
8 // Renvoie la valeur de l'élément coché (:checked) du groupe d'input "pays"
9 let pays = document.querySelector('input[name="pays"]:checked').value;
10
11 // Renvoie le champ texte "login" contenu dans une div avec la classe ".utilisateur"
12 const inputLogin = document.querySelector('div.utilisateur input[name="login"]');
```

Element.querySelectorAll()

Peut être appelée avec l'objet `document` ou un objet de type `Element`.

Retourne un **tableau** (NodeList) contenant une référence sur tous les éléments correspondant au **sélecteur CSS** - ou groupe de sélecteurs - spécifié(s).

```
1 // Retourne tous les paragraphes du document
2 const paras = document.querySelectorAll("p");
3
4 // Retourne tous les paragraphes présents dans une div avec la classe "article"
5 const parasArticle = document.querySelector("div.article > p");
6
7 // Retourne un tableau de tous les éléments correspondants à l'un des sélecteurs
8 // Sélectionne les div appartenant à la classe "note" OU "alert"
9 const notesEtAlertes = document.querySelectorAll("div.note, div.alert");
```

Modifier les contenus textes

Il existe deux propriétés pour récupérer ou modifier le contenu d'un élément HTML :

1. `innerHTML` : lecture ou modification au **format HTML**
2. `innerText` : lecture ou modification au **format texte brut**

La méthode, `insertAdjacentHTML()` permet elle d'ajouter du HTML à différents emplacement d'un élément.

innerHTML

Récupère ou définit le contenu HTML d'un élément et de ses descendants.

```
1  <p id="intro">
2    Je suis un <strong>joli</strong> paragraphe !
3  </p>
4
5  <script>
6    // Récupère le paragraphe #intro
7    const introPara = document.getElementById("intro");
8    // Récupère le contenu HTML du paragraphe
9    let contenu = introPara.innerHTML; //Je suis un <strong>joli</strong> para
10
11   // Remplacer le contenu HTML du paragraphe
12   introPara.innerHTML = "Je suis un <em>nouveau</em> paragraphe !";
13   // <p>Je suis un <em>nouveau</em> paragraphe !</p>
14
15   // Ajouter du contenu HTML à la fin du contenu existant avec +=
16   introPara.innerHTML += ' <a href="http://kode.ch">Lien</a>';
17   // <p>Je suis un <em>nouveau</em> paragraphe ! <a href="http://kode.ch">Li
18 </script>
```



innerText

Représente le contenu textuel, le rendu visuel d'un élément. Il fait donc abstraction des balises HTML.

Utilisé en lecture, il renvoie une approximation du texte que l'utilisateur ou utilisatrice obtiendrait s'il ou elle sélectionnait le contenu d'un élément avec le curseur, et le copiait dans le presse-papier.

```
1 <p id="intro">
2   Je suis un <strong>joli</strong> paragraphe !
3 </p>
4
5 <script>
6   // Récupère le paragraphe #intro
7   const introPara = document.getElementById("intro");
8
9   // Récupération du contenu avec innerText
10  console.log(introPara.innerText); // Je suis un joli paragraphe !
11
12  // Récupération du contenu avec innerHTML
13  console.log(introPara.innerHTML); // Je suis un <strong>joli</strong> para
14 </script>
```



133A-JS-InnerText

<https://codepen.io/fallinov/pen/qQVdXG?editors=0010>

insertAdjacentHTML(position, text);

Permet d'ajouter du `text` HTML à une `position` donnée autour ou à l'intérieur d'un `element` existant.

Il existe quatre `positions` :

- `'beforebegin'` : **Avant** l'`element` lui-même.

- 'afterbegin' : Juste à l'intérieur de l' `element` , **avant son premier enfant**.
- 'beforeend' : Juste à l'intérieur de l' `element` , **après son dernier enfant**.
- 'afterend' : **Après** `element` lui-même.

Exemple de ces positions pour le paragraphe `#intro`

```

1 <h2>Mon titre</h2>
2 <p>Un peu de texte</p>
3 <!-- beforebegin -->
4 <p id='intro'>
5   <!-- afterbegin -->
6   Et encore un peu de texte
7   <!-- beforeend -->
8 </p>
9 <!-- afterend -->
10 <p>Et encore et toujours du texte</p>
11 <h3>Mon sous-titre</h3>

```

Exemple

```

1 <ul id="liste1">
2   <li>élément de #liste1</li>
3 </ul>
4
5 <ul id="liste2">
6   <li>élément de #liste2</li>
7 </ul>
8
9 <script>
10 // Ajouter un nouveau contenu entre #liste1 et #liste2
11 const liste2 = document.getElementById('liste2');
12
13 // Ajout de <p> juste avant et juste après #liste2
14 liste2.insertAdjacentHTML('beforebegin', '<p>Juste avant #liste2</p>');
15 liste2.insertAdjacentHTML('afterend', '<p>Juste après #liste2</p>');
16
17 // Ajout de <li> comme premier et dernier fils de la #liste2
18 liste2.insertAdjacentHTML('afterbegin', '<li>Nouveau premier fils de #liste2</li>');
19 liste2.insertAdjacentHTML('beforeend', '<li>Nouveau dernier fils de #liste2</li>');
20 </script>

```



133A-JS-insertAdjacentHTML

[https://codepen.io/fallinov/pen/qQVdPz?](https://codepen.io/fallinov/pen/qQVdPz?editors=1000)
editors=1000

Modifier le style CSS

Element.style

La propriété `style` d'un élément représente son attribut HTML `style="color:red;"`. Elle représente donc la **déclaration de style en-ligne** qui a la **priorité la plus haute** dans la cascade CSS.

Cependant, elle n'est **pas utile pour connaître le style de l'élément** en général, puisqu'elle ne représente que les déclarations CSS définies dans l'attribut `style` de l'élément, et pas celles qui viennent d'autres règles de style.

Pour obtenir les valeurs de toutes les propriétés CSS pour un élément, il faut utiliser `window.getComputedStyle(element)`.

Pour ajouter ou modifier une déclaration CSS dans l'attribut `style` d'un élément on écrira

```
Element.style.propriétéCSS = "valeur"
```

En JavaScript deux règles importantes concernant le CSS :

- les **valeurs sont toujours des chaînes de caractères**

```
Element.style.padding = "4px" .
```

- les traits d'union `-` des **propriétés CSS composées de plusieurs mots-clés** comme `border-color`, sont remplacés par une **camélisation** `borderColor`.

border-color ⇒ **borderColor**

Camélisation des propriétés CSS

Ci-après, quelques exemples de déclaration CSS et leur équivalence en JavaScript:

Déclaration CSS

JavaScript

color: #2ecc71;

Element.style.color = "#2ecc71";

font-size: 2em;

Element.style.fontSize = "2em";

background-color: red;

Element.style.backgroundColor = "red";

border-top-width : 2px;

Element.style.borderTopWidth = "2px";

color: #333;

Element.style.color = "#333";

Exemple

```
1  const intros = document.getElementsByClassName("intro");
2
3  for (let i = 0; i < intros.length; i = i + 1) {
4      intros[i].style.fontSize = '1.5em';
5      intros[i].style.backgroundColor = 'lime';
6  }
```

window.getComputedStyle(element)

La méthode `window.getComputedStyle()` retourne un objet contenant la valeur calculée finale de toutes les propriétés CSS d'un élément.



L'objet retourné est en **lecture seule**.

Exemple

```
1  // Récupère #intro
2  const intro = document.getElementById('intro');
```

```
3 // Récupère le style CSS de #intro
4 let styleIntro = window.getComputedStyle(intro);
5 // Affiche la valeur de la propriété CSS top de #intro
6 console.log( styleIntro.getPropertyValue('top') ); // "40px"
```



133A - window.getComputedStyle()

<https://codepen.io/fallinov/pen/EdMxzL?editors=0011>

Modifier les attributs

Les attributs standard HTML sont accessible comme propriété de l'objet représentant l'élément HTML. On peut donc y accéder en lecture et écriture en écrivant :

```
elementHTML.nomAttribut
```



Pour modifier les attributs HTML non-standard utiliser `setAttribute()`

```
1 <h1>Modifier les attributs d'un élément HTML</h1>
2
3 <a id="delemont">Visiter Delémont</a>
4 <a id="porrentruy">Visiter Porrentruy</a>
5
6 <script>
7   const delemont = document.getElementById("delemont");
8   delemont.href = "http://www.delemont.ch";
9   delemont.target = "_blank";
10
11   // Avec la méthode setAttribute
12   const porrentruy = document.getElementById("porrentruy");
13   porrentruy.setAttribute("href", "http://www.porrentruy.ch");
14   porrentruy.setAttribute("target", "_blank");
15 </script>
```



133A - modifier les attributs

<https://codepen.io/fallinov/pen/rNBLQZB?editors=0010>

Modifier les classes CSS

```
1 // Récupère l'élément #menu
2 const menu = document.getElementById('menu');
3
4 // Supprime la class rouge de #menu si présente
5 menu.classList.remove('rouge');
6
7 // Ajoute la class vert à #menu si non présente
8 menu.classList.add('vert');
9
10 // Ajoute ou retire plusieurs classes
11 menu.classList.add('jaune', 'bleu');
12 menu.classList.remove('jaune', 'bleu');
13
14 /* Alternance :
15     Si #menu a la classe .rouge toggle('rouge') la retire
16     Si #menu n'a pas la classe .rouge toggle('rouge') l'ajoute */
17 menu.classList.toggle('rouge');
18
19 // Retourne true si #menu a la classe .rouge, false s'il ne l'a pas
20 menu.classList.contains('rouge');
```

Créer des éléments

Ajouter un élément enfant à la fin d'un élément existant

Pour ajouter un élément comme dernier fils d'un élément existant il faut :

1. Créer un nouvel élément : `createElement("nomTagHTML")`
2. Créer un nœud texte : `createTextNode("chaîne de caractères")`
3. Attacher le nœud texte au nouvel élément : `appendChild(nœudTexte)`
4. Récupérer un élément existant du DOM : voir chapitre [Accéder aux éléments de la DOM](#)
5. Attacher le nouvel élément à l'élément existant du DOM : `appendChild(element)`

Exemple : Ajouter un élément à la fin d'une liste

Voici comment ajouter le nouvel élément `2kg de Pain` **à la fin** de la liste `#fondue` .

```
1 <ul id="fondue">
2   <li>2kg de Fromage</li>
3   <li>1L de Kirsh</li>
4 </ul>
5
6 <script>
7 // 1. Création du nouvel élément <li>
8 const newLi = document.createElement('li');
9
10 // 2. Création du nœud texte
11 const newLiTexte = document.createTextNode("2kg de Pain");
12
13 // 3. Ajout du texte au <li>
14 newLi.appendChild(newLiTexte);
15
16 // 4. Récupération de la liste
17 const listeFondue = document.getElementById('fondue');
18
19 // 5. Ajoute le nouvel élément <li> à la fin de la liste
20 listeFondue.appendChild(newLi);
21 </script>
```

Résultat

```
1 <ul id="fondue">
2   <li>2kg de Fromage</li>
3   <li>1L de Kirsh</li>
4   <li>2kg de Pain</li>
5 </ul>
```

Ajouter un nouvel élément avant un élément enfant existant

L'exemple précédent nous a montré comment ajouter un nouvel élément enfant à la fin d'un élément existant avec `appendChild()` .

Il existe une autre méthode pour ajouter des éléments : `element.insertBefore()`

```
elementParent.insertBefore(nouvelElement, elementEnfantExistant);
```

Cette méthode permet d'ajouter à un élément existant `elementParent` un élément enfant `nouvelElement` juste avant l'élément enfant spécifié `elementEnfantExistant` .

Exemple : Ajouter un élément au début d'une liste

Voici comment ajouter le nouvel élément `2kg de Pain` **au début** de la liste `#fondue` .

```
1 <ul id="fondue">
2   <li>2kg de Fromage</li>
3   <li>1L de Kirsh</li>
4 </ul>
5
6 <script>
7 // 1. Création du nouvel élément <li>
```

```
8  const newLi = document.createElement('li');
9
10 // 2. Création du nœud texte
11 const newLiTexte = document.createTextNode("2kg de Pain");
12
13 // 3. Ajout du texte au <li>
14 newLi.appendChild(newLiTexte);
15
16 // 4. Récupération d'un 1er élément <li> de la liste actuelle existant du
17 const premierLi = document.querySelector('#fondue li:first-child');
18
19 // 3. Récupération du parent de premierLi
20 const parentLi = premierLi.parentNode;
21
22 // 6. Ajout de newLi avant premierLi
23 parentLi.insertBefore(newLi, premierLi);
24 </script>
```

Résultat

```
1  <ul id="fondue">
2    <li>2kg de Pain</li>
3    <li>2kg de Fromage</li>
4    <li>1L de Kirsh</li>
5  </ul>
```

Supprimer, remplacer et cloner

Supprimer un élément

```
1 <div>
2   <h1>Un Titre</h1>
3   <p>Petit paragraphe<p>
4 </div>
5
6 <script>
7 // Récupération du 1er paragraphe de la 1re div du document
8 const p1 = document.querySelector("div p");
9 // Suppression du 1er paragraphe
10 p1.remove();
11 </script>
```



133A-JS-remove - JSFiddle - Code Playground

<https://jsfiddle.net/fallinov/zybnhj6c/>

Supprimer un élément fils

```
1 <div>
2   <h1>Un Titre</h1>
3   <p>Petit paragraphe<p>
4 </div>
5
6 <script>
7 // Récupération de la 1re div du document
8 const div = document.querySelector("div");
9 // Récupération du 1er <p> de la DIV
10 const p1 = div.querySelector("p");
11 // Suppression du 1er paragraphe
12 div.removeChild(p1);
13 </script>
```




Remplacer un élément

```
1 <div>
2   <h1>Un Titre</h1>
3   <p>Petit paragraphe<p>
4 </div>
5
6 <script>
7   // Récupération de la 1re div du document
8   const div = document.querySelector("div");
9   // Récupération du 1er <p> de la DIV
10  const ancienPara = div.querySelector("p");
11  // Création d'un nouveau <p>
12  const nouveauPara = document.createElement("p");
13  // Modification du texte du nouveau <p>
14  nouveauPara.innerText = "Nouveau paragraphe";
15  // Remplace l'ancien <p> par le nouveau
16  div.replaceChild(nouveauPara, ancienPara);
17 </script>
```



Cloner un élément

```
1 <ul id="liste1">
2   <li>Fromage</li>
3   <li>Thé</li>
4 </ul>
5
6 <ul id="liste2">
7   <li>Eau</li>
```

```
8     <li>Sucre</li>
9 </ul>
10
11 <script>
12 // Récupération du dernier fils de #liste1 <li>Thé</li>
13 const dernierFilsListe1 = document.querySelector("#liste1 :last-child");
14 // Clone, copie, le dernier fils et son contenu, sa descendance
15 const cloneDernierFils = dernierFilsListe1.cloneNode(true);
16 // Ajoute le clone à la fin de #liste2
17 document.getElementById("liste2").appendChild(cloneDernierFils);
18 </script>
```



133A - cloneNode()

<https://codepen.io/fallinov/pen/LgaMja?editors=1111>

Événements

Les événements permettent de déclencher une fonction pour une action spécifique, comme par exemple le clic ou le survol d'un élément, le chargement du document HTML ou encore l'envoi d'un formulaire.

Principaux événements du DOM

Événement DOM	Description
<code>click</code>	Bouton de la souris enfoncé puis relâché sur un élément.
<code>dblclick</code>	Deux fois l'événement <code>click</code>
<code>mouseover</code>	Souris au-dessus d'un élément.
<code>mouseout</code>	Souris sort d'un élément.
<code>mousedown</code>	Bouton de la souris enfoncé, pas relâché, sur un élément.
<code>mouseup</code>	Bouton de la souris relâché sur un élément.
<code>mousemove</code>	Souris en mouvement au-dessus d'un élément.
<code>keydown</code>	Touche clavier enfoncée, pas relâchée, sur un élément.
<code>keyup</code>	Touche clavier relâchée sur un élément.
<code>keypress</code>	Touche clavier enfoncée et relâchée sur un élément.
<code>focus</code>	L'élément reçoit, gagne, le focus. Quand un objet devient l'élément actif du document.
<code>blur</code>	Élément perd le focus.
<code>change</code>	Changement de a valeur d'un élément de formulaire.

`select`

Sélection du texte d'un élément, mis en surbrillance.

`submit`

Envoi d'un formulaire

`reset`Réinitialisation d'un formulaire



Liste complète des événements :

https://www.w3schools.com/jsref/dom_obj_event.asp

Affecter une fonction à un événement

Il existe différentes manières d'affecter une fonction à l'événement d'un objet.

- Utiliser les **gestionnaires d'événements "on-event"** 👎
- Créer des écouteurs d'événement (listener) avec **la méthode** `addEventListener()` 👍

**Le meilleur moyen est souvent `addEventListener()`**

Avec la méthode "on-event", chaque objet ne peut avoir qu'un seul gestionnaire d'événement pour un événement donné. C'est pourquoi `addEventListener()` est souvent le meilleur moyen d'être averti des événements.

On-event

Les gestionnaires d'événements "on-event" sont nommés selon l'événement lié : `onclick` , `onkeypress` , `onfocus` , `onsubmit` , etc.



Liste des gestionnaires d'événements :

https://www.w3schools.com/tags/ref_eventattributes.asp

On peut spécifier un "on-event" pour un événement particulier de différentes manières :

- Avec un attribut HTML : `<button onclick="bonjour()">`
- En utilisant la propriété correspondante en JavaScript : `Element.onclick = bonjour;`

❗ En JavaScript, afin d'affecter la fonction `bonjour()` et non son résultat, on n'ajoute pas les parenthèses après le nom de la fonction.

- `Element.onclick = bonjour;` affecte la fonction `bonjour()` .
- `Element.onclick = bonjour();` affecte le résultat de la fonction `bonjour()` .

```
1  function citationLeia() {
2      alert("Plutôt embrasser un Wookie");
3  }
4
5  // Affecte la fonction citationLeia() au click du bouton
6  document.querySelector('button').onclick = citationLeia;
7
8  // Variante avec fonction anonyme
9  document.querySelector('button').onclick = function() {
10     alert("Plutôt embrasser un Wookie");
11 };
12
13 // Variante avec fonction fléchée (arrow function)
14 document.querySelector('button').onclick = () => alert("Plutôt embrasser u
```

addEventListener()

 Liste des événements JavaScript :

https://www.w3schools.com/jsref/dom_obj_event.asp

La méthode `addEventListener()` permet de définir une fonction à appeler chaque fois que l'événement spécifié est détecté sur l'élément ciblé.

```
ElementCible.addEventListener("nomEvenement", nomFonction);
```

```
1  const newElement = document.getElementsByTagName('h1');
2
3  newElement.onclick = function() {
4    console.log('clicked');
5  };
6
7  let logEventType = function(e) {
8    console.log('event type:', e.type);
9  };
10
11 newElement.addEventListener('focus', logEventType, false);
12 newElement.removeEventListener('focus', logEventType, false);
13
14 window.onload = function() {
15   console.log('Im loaded');
16 };
```

Ajouter un événement à une liste d'éléments

main.js

```
1  const boutons = document.querySelectorAll("button");
2
3  for (let bouton of boutons) {
4    bouton.addEventListener("click", function(event) {
5      bouton.classList.toggle("rouge");
6    });
7  }
```

index.html

```
1 <button>Bouton 1</button>
2 <button>Bouton 3</button>
3 <button>Bouton 3</button>
```

styles.css

```
1 button {
2     cursor: pointer;
3     color: #7f8c8d;
4     font-weight: bold;
5     background-color: #ecf0f1;
6     padding: 1em 2em;
7     border: 2px solid #7f8c8d;
8 }
9
10 button.rouge {
11     border-color: #c0392b;
12     background-color: #e74c3c;
13     color: #ecf0f1;
14 }
```



133A - Ajouter un événement click à une liste d'éléments

<https://codepen.io/fallinov/pen/WaPXEQ/>

L'objet event

Un objet `event` est automatiquement passé comme premier paramètre de la fonction affectée à un événement. Pour le récupérer il suffit d'ajouter un paramètre à la fonction liée.

Le nom de ce paramètre est libre mais on le nomme régulièrement `event` ou plus simplement `e`.

```
1 <button>Clique moi !</button>
2
3 <script>
4 // Récupère le 1er boutons du document
5 const bouton = document.querySelector("button");
6 // Ajoute événement click avec une fonction avec paramètre event
7 bouton.addEventListener("click", function (event) {
8   // Affiche le type d'événement envoyé
9   alert(event.type); // click
10 });
11 </script>
```



133A - event object

<https://codepen.io/fallinov/pen/PyLVjJ>

Récupérer la cible d'un événement

On appelle "cible" l'objet ou l'élément qui a envoyé l'événement. Pour récupérer la cible on utilise la propriété `target` de l'événement.

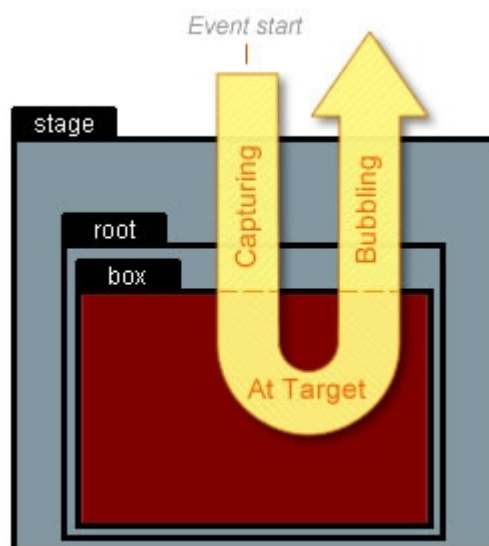
```
1 <button>Clique moi !</button>
2
3 <script>
4 // Récupère le 1er boutons du document
5 const bouton = document.querySelector("button");
6 // Ajoute événement click avec une fonction avec paramètre event
7 bouton.addEventListener("click", function (event) {
8   // Récupère l'élément qui a envoyé l'événement, la cible
9   let cible = event.target;
10  // Modifie la taille du texte de la cible
11  cible.style.fontSize = "2em";
12  // Affiche le contenu texte de la cible
13  alert(cible.innerText); // Clique moi !
14 });
15 </script>
```




Bubbling & Capturing

Capture ? Bouillonnement ? De quoi parle-t-on ?

Ces deux phases sont deux étapes distinctes de l'exécution d'un événement. La première, la **capture** (*capture* en anglais), s'exécute avant le déclenchement de l'événement, tandis que la deuxième, le **bouillonnement** (*bubbling* en anglais), s'exécute après que l'événement a été déclenché. Toutes deux permettent de définir le sens de propagation des événements.



```
1 <div id="div1">
2   <p id="p1">I am Bubbling</p>
3 </div><br>
4
5 <div id="div2">
6   <p id="p2">I am Capturing.</p>
7 </div>
8
9 <script>
10 document.getElementById("p1").addEventListener("click", function() {
11     alert("You clicked the P element!");
12 }, false);
```

```
13
14 document.getElementById("div1").addEventListener("click", function() {
15     alert("You clicked the DIV element!");
16 }, false);
17
18 document.getElementById("p2").addEventListener("click", function() {
19     alert("You clicked the P element!");
20 }, true);
21
22 document.getElementById("div2").addEventListener("click", function() {
23     alert("You clicked the DIV element!");
24 }, true);
25 </script>
```



133A - bubbling & capturing

<https://codepen.io/fallinov/pen/zmbeev?editors=1111>

A lire...

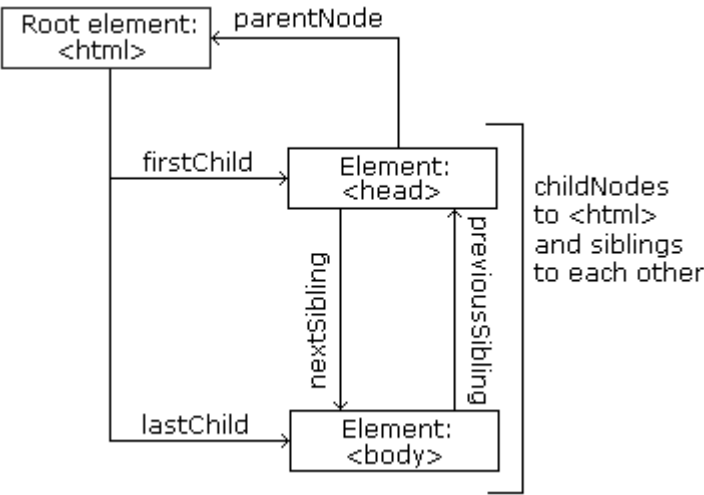


Creating and triggering events

https://developer.mozilla.org/en-US/docs/Web/Guide/Events/Creating_and_triggering_events

Naviguer dans le DOM

"Naviguer dans la DOM", représente l'action de se déplacer, ou récupérer un noeud parent, enfant ou adjacent.



Propriétés de navigations DOM

Tableau des propriétés

Action	Propriétés des noeud	Retour
Récupérer les fils	<code>node.childNodes</code> <code>element.children</code>	<code>NodeList</code> <i>tableau de noeuds</i> <code>HTMLCollection</code> <i>tableau d'éléments</i>
Premier fils	<code>node.firstChild</code> <code>element.firstChild</code>	<code>Node</code> <code>Element</code>
Dernier fils	<code>node.lastChild</code> <code>element.lastElementChild</code>	<code>Node</code> <code>Element</code>
Frère suivant	<code>node.nextSibling</code>	<code>Node</code>
Frère précédent	<code>node.previousSibling</code>	<code>Node</code>

Récupérer le
parent

`node.parentNode`

`Node`

Comme on peut le voir, certaines actions sont réalisables avec deux propriétés différentes.

Il existe deux "familles" de propriétés pour naviguer dans la DOM :

- Les propriétés de type `Node` , pour naviguer dans **tous les types de noeuds** :
éléments, textes ou commentaires
- Les propriétés de type `Element` , pour naviguer **uniquement dans les éléments**

Dans l'exemple ci après, on récupère le premier fils de la `div` avec la propriété `firstChild` , avec la propriété `firstElementChild` .

- La propriété `firstChild` retourne le premier noeud peu importe son type.
Dans l'exemple le premier noeud est un noeud texte contenant `"Bonjour le"` .
- La propriété `firstElementChild` retourne le premier noeud de type élément.
Dans l'exemple le premier élément de la div est `monde` .

```
1 <div>Bonjour le <strong>monde</strong>!</div>
2
3 <script>
4 // Récupère la première <div> du document
5 const div = document.querySelector("div");
6
7 // Affiche le premier fils avec firstChild
8 console.log( div.firstChild ); // "Bonjour le"
9
10 // Affiche le premier fils de type Element avec firstElementChild
11 console.log( div.firstElementChild ); // "<strong>monde</strong>"
12 </script>
```

Propriétés de type `Node`

```
1 <div>Bonjour <strong>le <em>monde</em></strong>!</div>
2
```

```

3  <script>
4  // Récupère le premier élément strong du document
5  const strongElement = document.querySelector("strong");
6
7  // Noeud parent
8  monElement.parentNode; // <div>
9
10 // Tous les fils (tableau de noeuds)
11 monElement.childNodes; // ["le ", <em>]
12
13 // Premier fils
14 monElement.firstChild; // "le "
15
16 // Dernier fils
17 monElement.lastChild; // <em>
18
19 // Frère suivant
20 monElement.nextSibling; // "!"
21
22 // Frère précédent
23 monElement.previousSibling; // "Bonjour "
24 </script>

```

Propriétés de type **Element**

```

1  <div id="animaux">
2    <h1>Animaux</h1>
3    <ul>
4      <li>  </li>
5      <li>  </li>
6    </ul>
7  </div>
8
9  <script>
10 // Récupère la première liste non triée du document
11 const listeAnimaux = document.querySelector("ul");
12
13 // Noeud parent
14 listeAnimaux.parentNode; // <div id="animaux">...</div>
15
16 // Tous les fils (tableau de noeuds)
17 listeAnimaux.children; // [<li>  </li>, <li>  </li>]
18
19 // Premier fils
20 listeAnimaux.firstElementChild; // <li>  </li>
21

```

```
22 // Dernier fils
23 listeAnimaux.lastElementChild; // <li> </li>
24 </script>
```

Formulaires

Envoyer des formulaires

Envoyer et réinitialiser un formulaire

Pour envoyer un formulaire on utilise la méthode `submit()` et `reset()` pour le réinitialiser.

```
1 // Récupère le 1er formulaire du document
2 const formulaire = document.querySelector('form');
3
4 // Envoyer un formulaire
5 formulaire.submit();
6
7 // Réinitialiser un formulaire
8 formulaire.reset();
```

Événement `submit`

L'événement `submit` permet de déclencher une fonction lors de l'envoi du formulaire.

```
1 const formulaire = document.querySelector('form');
2
3 formulaire.addEventListener('submit', function(){
4     console.log("Formulaire envoyé !");
5 });
```

Une fois la fonction terminée le formulaire sera envoyé.

Stopper l'envoi du formulaire

Si l'on veut désactiver, stopper l'envoi du formulaire il faut utiliser la méthode `preventDefault()` de l'événement.


```
1  const formulaire = document.querySelector('form');
2
3  // Ne pas oublier d'ajouter un paramètre à la fonction pour récupérer l'év
4  formulaire.addEventListener('submit', function(event){
5      event.preventDefault(); // Stoppe l'envoi du formulaire
6      console.log("Formulaire envoyé !");
7  });
```

Exemple

```
1  <form action="https://kode.ch/getpost/" method="post">
2      <label for="nom">Votre nom</label>
3      <input type="text" id="nom" name="nom">
4      <button>Envoyer</button>
5  </form>
6
7  <script>
8      // 1er formulaire du document
9      const formulaire = document.querySelector("form");
10     // Champ texte nom
11     const inputNom = document.getElementById("nom");
12     // Événement submit => Lors de l'envoi du formulaire
13     formulaire.addEventListener("submit", function(event) {
14         // Désactive l'envoi du formulaire
15         event.preventDefault();
16
17         // Si utilisateur n'a pas saisi de nom
18         if (inputNom.value === "") {
19             alert("Entrez votre nom !");
20             return; // Sors de la fonction
21         }
22
23         // Envoie le formulaire
24         formulaire.submit();
25     });
26
27 </script>
```



133A-JS-Premier formulaire

<https://codepen.io/fallinov/pen/yQPywX?editors=0010>

Récupérer et modifier la valeur des champs

Champs de saisie

Propriété `value`

Pour récupérer ou modifier la valeur entrée par le visiteur dans un champs de saisie texte (`input` , `password` , `textarea` , `hidden` , `email`) on utilise la propriété `.value` .

```
1 let texte = monElement.value;  
2 monElement.value = "toto";
```

Liste déroulantes

Propriété `value`

Pour récupérer la valeur de l'option sélectionnée d'une liste, on utilise la propriété `.value` .

```
monElementSelect.value;
```

Selecteur CSS `:checked`

Le sélecteur `:checked` vous permet de récupérer l'**option actuellement sélectionnée** avec `querySelector` .

Exemple récupérer le contenu texte de l'option `selected` :

```
1 <select name="pays" id="pays">
```

```

2   <option value="">-- Sélectionnez un pays --</option>
3   <option value="FR">France</option>
4   <option selected value="IT">Italie</option>
5   <option value="CH">Suisse</option>
6 </select>
7
8 <script>
9   const liste = document.getElementById("pays");
10  const optionSelectionnee = liste.querySelector("option:checked");
11
12  console.log(optionSelectionnee.innerText); // Italie
13 </script>

```

Événement `change`

L'événement `change` est souvent associé aux listes. Il se déclenche lorsque le visiteur sélectionne une autre option dans la liste.

```
monElementListe.addEventListener("change", function() {...});
```

Exemple

```

1  <select name="pays" id="pays">
2    <option selected value="">-- Sélectionnez un pays --</option>
3    <option value="FR">France</option>
4    <option value="IT">Italie</option>
5    <option value="CH">Suisse</option>
6 </select>
7
8 <div>
9   Code du pays : <span class="code"></span>
10 </div>
11
12 <script>
13 // Récupère la liste déroulante #pays et le span .code
14 const listePays = document.getElementById("pays");
15 const codeSelectionne = document.querySelector("span.code");
16
17 // Sur changement de la valeur de la liste déroulante
18 listePays.addEventListener("change", function() {

```

```
19 // Récupère la valeur de l'option sélectionnée
20 let codePays = listePays.value;
21 // Modifie le contenu texte du span .code
22 codeSelectionne.innerText = codePays;
23 });
24 </script>
```



133A-JS-Select change

[https://codepen.io/fallinov/pen/jQaPVL?](https://codepen.io/fallinov/pen/jQaPVL?editors=1000)
editors=1000

Cases à cocher

Propriété checked

La propriété `checked` vous permet de savoir si une case est cochée `true` ou non `false`

```
monElement.checked; // Retourne true ou false
```

Exemple

```
1 <form action="https://kode.ch/getpost/" method="post">
2   <input type="checkbox" id="copie" name="copie">
3   <label for="copie">Recevoir une copie</label>
4   <button>Envoyer</button>
5 </form>
6
7 <script>
8 // 1er formulaire du document
9 const formulaire = document.querySelector("form");
10 // Case à cocher "copie"
11 const chkCopie = document.getElementById("copie");
12
13 // Événement submit => Lors de l'envoi du formulaire
14 formulaire.addEventListener("submit", function(event) {
15   // Si utilisateur n'a pas saisi de nom
```

```
16     if (chkCopie.checked === true) {
17         alert("Message envoyé AVEC copie !");
18     } else {
19         alert("Message envoyé SANS copie !");
20     }
21 });
22 </script>
```



133A-JS-Cases à cocher

<https://codepen.io/fallinov/pen/zMPGvL?editors=0010>

Groupe de cases à cocher

Pour récupérer les cases cochées d'un groupe, la meilleure méthode est d'utiliser `querySelector` et la puissance des sélecteurs CSS, pour récupérer toutes les cases cochées `:checked` du groupe `[name="nomGroupe"]`.

```
1 let casesCochées = document.querySelectorAll(
2     'input[name="groupeCases[]"]:checked'
3 );
```



La variable qui contient le résultat du `querySelectorAll()` n'est pas "dynamique", les nouvelles cases cochées ne s'y ajouteront pas automatiquement.

Il ne faut donc rappeler `querySelectorAll()` pour mettre à jour le contenu de la variable.

Exemple

```

1  <form action="https://kode.ch/getpost/" method="post">
2    <input type="checkbox" name="couleurs[]" id="rouge" value="rouge">
3    <label for="rouge">Rouge</label>
4
5    <input type="checkbox" name="couleurs[]" id="vert" value="vert">
6    <label for="vert">Vert</label>
7
8    <input type="checkbox" name="couleurs[]" id="bleu" value="bleu">
9    <label for="bleu">Bleu</label>
10
11    <button>Envoyer</button>
12 </form>
13
14 <script>
15 // 1er formulaire du document
16 const formulaire = document.querySelector("form");
17
18 // Événement submit => Lors de l'envoi du formulaire
19 formulaire.addEventListener("submit", function(event) {
20     event.preventDefault();
21
22     // Cases cochée dans le groupe couleurs[]
23     const couleursCochées = document.querySelectorAll(
24         'input[name="couleurs[]"]:checked'
25     );
26
27     //Récupère la valeur des éléments cochés
28     for (let couleur of couleursCochées) {
29         alert(couleur.value);
30     }
31 });
32 </script>

```



133A-JS-Groupe de cases à cocher

<https://codepen.io/fallinov/pen/eQeNYV?editors=0011>

Groupe de boutons radios

Pour récupérer la valeur du radio sélectionné dans un groupe, la meilleure méthode est d'utiliser `querySelector` et la puissance des sélecteurs CSS, pour récupérer le premier

radio coché `:checked` du groupe `[name="nomGroupe"]` .

```
1 // Récupère la valeur du radio coché dans le groupe "couleur"
2 document.querySelector('[name="couleur"]:checked').value;
```

Exemple

```
1 <form action="https://kode.ch/getpost/" method="post">
2   <input type="radio" name="genre" id="h" value="Homme">
3   <label for="h">Homme</label>
4
5   <input type="radio" name="genre" id="f" value="Femme">
6   <label for="f">Femme</label>
7
8   <button>Envoyer</button>
9 </form>
10
11 <script>
12 // 1er formulaire du document
13 const formulaire = document.querySelector("form");
14
15 // Événement submit => Lors de l'envoi du formulaire
16 formulaire.addEventListener("submit", function(event) {
17   // Désactive l'envoi du formulaire
18   event.preventDefault();
19
20   // Radio coché dans le groupe genre
21   const genre = document.querySelector('[name="genre"]:checked');
22
23   // Test si un genre est coché
24   if (genre === null) {
25     alert("Sélectionner un genre !");
26     return;
27   }
28
29   alert(genre.value);
30 });
31 </script>
```




133A - JS - Form Option group

[https://codepen.io/fallinov/pen/aQzRPy?
editors=0010](https://codepen.io/fallinov/pen/aQzRPy?editors=0010)

Valider les saisies utilisateurs

Ci-après un exemple classique de validation de formulaire avec création d'un message d'erreur.

HTML

```
1 <ul class="message"></ul>
2 <form action="https://kode.ch/getpost/" method="post">
3   <ul>
4     <li>
5       <label for="nom">Votre nom</label>
6       <input type="text" id="nom" name="nom">
7     </li>
8     <li>
9       <label for="age">Votre age</label>
10      <input type="text" id="age" name="age">
11    </li>
12    <li>
13      <button type="submit">Envoyer</button>
14    </li>
15  </ul>
16 </form>
```

```

1  /**
2   * Valide le nom et l'âge d'une personne et retourne un tableau d'er
3   * @return {Array} Tableau de messages d'erreur
4   */
5  function validerPersonne(nom, age) {
6      // Initialisation du tableau des erreurs
7      const erreurs = [];
8
9      //Supprime les espaces en début et fin de chaine
10     nom = nom.trim();
11
12     //Converti l'age en entier
13     age = parseInt(age);
14
15     // Si le nom vide
16     if (nom === "") {
17         erreurs.push("Entrez un nom !");
18     }
19
20     // Si l'âge n'est pas un nombre entier compris entre 0 et 120
21     if (Number.isNaN(age) || age < 1 || age > 119) {
22         erreurs.push("Entrez un age valide !");
23     }
24
25     return erreurs;
26 }
27
28 /**
29 * Ajoute le contenu d'un tableau à la fin d'une liste HTML
30 * @param {HTMLElement} eleListe - Liste HTML (ol ou ul) à remplir
31 * @param {Array} erreurs - tableau de String
32 */
33 function ajouterFinListe(eleListe, erreurs) {
34     // Parcours les messages d'erreur
35     for (message of erreurs) {
36         // Ajoute un li au contenu de la liste
37         eleListe.innerHTML += "<li>" + message.toString() + "</li>";
38     }
39 }
40
41 // Récupération du formulaire et de la liste message
42 const eleFormulaire = document.querySelector("form");
43 const eleMessage = document.querySelector("ul.message");
44
45 // Événement submit => Lors de l'envoi du formulaire
46 eleFormulaire.addEventListener("submit", function(event) {
47     // Désactive l'envoi du formulaire
48     event.preventDefault();

```

```
49
50 // Récupère les champs nom et age
51 const txtNom = document.getElementById("nom");
52 const txtAge = document.getElementById("age");
53
54 // Supprime les anciens messages d'erreur
55 eleMessage.innerHTML = "";
56
57 // Validation des données
58 let erreurs = validerPersonne(txtNom.value, txtAge.value);
59
60 // Si il y a des erreurs
61 if (erreurs.length > 0) {
62     // Ajoute les erreurs à la fin de ul.message
63     ajouterFinListe(eleMessage, erreurs);
64 } else {
65     // Envoi du formulaire
66     eleFormulaire.submit();
67 }
68 });
```

```
1  body {
2      font-family: "Trebuchet MS", Helvetica, sans-serif;
3  }
4
5  ul.message {
6      color: #ecf0f1;
7      background-color: #e74c3c;
8  }
9
10 ul.message li {
11     padding: .5em 0;
12 }
13
14 form ul {
15     list-style-type: none;
16     padding: 0;
17 }
18
19 form ul li {
20     padding: 0 0 1em 0;
21 }
22
23 form ul li label {
24     display: block;
25     font-weight: bold;
26 }
27
```



JavaScript Moderne

Transpiler



Babel · The compiler for next generation JavaScript

<https://babeljs.io/>

Fonctions fléchées (Arrow Function)

Les bases

Les fonctions fléchées ont été introduites dans ES6. Elles permettent d'écrire une fonction de manière raccourcie :

```
1 // Fonction régulière
2 hello = function() {
3   return "Hello World!";
4 }
5
6 // Avec une fonction fléchée
7 hello = () => {
8   return "Hello World!";
9 }
10
11 // On peut faire encore plus court !
12 // Si la fonction ne comporte qu'une seule déclaration et renvoie une valeur
13 // vous pouvez supprimer les parenthèses et le mot-clé return :
14 hello = () => "Hello World!";
15
16 // Même chose avec un paramètre
17 hello = (val) => "Hello " + val;
18
19 // Si il n'y a qu'un seul paramètre, on peut aussi supprimer les parenthèses
20 hello = val => "Hello " + val;
```

Qu'en est-il de **this** !

Le traitement de `this` n'est pas le même dans les fonctions fléchées que dans les fonctions régulières.



En bref, les fonctions fléchées n'injectent pas l'objet `this`, donc ne l'utilisez pas !

Dans les **fonctions régulières**, `this` représente l'**objet qui appelle la fonction**, qui peut être la fenêtre, le document, un bouton ou autre.

Dans les **fonctions fléchées**, `this` représente toujours l'**objet qui a créé la fonction fléchée**, s'il y en a eu un.

Comme le présente l'exemple ci-après, l'objet `this` est `undefined` dans la fonction fléchée.

```
1 // Fonction régulière, this représente l'objet qui a appelé la fonction
2 hello1 = function() {
3   alert(this.id)
4 }
5 document.getElementById("btn1").addEventListener("click", hello1);
6 // Résultat: btn2
7
8 // Fonction fléchée, this n'existe pas
9 hello2 = () => {
10  alert(this.id)
11 }
12 document.getElementById("btn2").addEventListener("click", hello2);
13 // Résultat: undefined
```



133 - Fonctions fléchées - this

<https://codepen.io/fallinov/embed/VwLQPyj?height=300&theme-id=26660&default-tab=js,result>

Template Literals

Les **Template literals** permettent d'écrire des **chaines de caractères multilignes** contenant des **expressions**.

- Ces chaines spéciales sont délimitées par des **accents graves** ``ma chaine``.
- Les expressions commencent par un `$` et sont délimitées par des accolades :
`${expression}`

```
1 let nom = 'Skywlaker';
2 let prenom = 'Luc'
3 let message = `Salut ${prenom} ${nom} !`;
4
5 alert(message); // Salut Luc Skywlaker !
6
7 let ficheClient = `

133A-JS-Template Literals



https://codepen.io/fallinov/pen/NEXGGX?editors=0010


```

Ensembles

Les ensembles `Set` est un nouveau type d'objet arrivé avec ES6 (ES2015), qui permet de créer des collections de valeurs uniques.

Voici un exemple simple montrant un ensemble de base et quelques-unes des méthodes disponibles comme `add`, `size`, `has`, `forEach`, `delete` et `clear`.

```
1  let animals = new Set();
2
3  animals.add(' ');
4  animals.add(' ');
5  animals.add(' ');
6  animals.add(' ');
7  console.log(animals.size); // 4
8  animals.add(' ');
9  console.log(animals.size); // 4
10
11 console.log(animals.has(' ')); // true
12 animals.delete(' ');
13 console.log(animals.has(' ')); // false
14
15 animals.forEach(animal => {
16   console.log(`Hey ${animal}!`);
17 });
18
19 // Hey  !
20 // Hey  !
21 // Hey  !
22
23 animals.clear();
24 console.log(animals.size); // 0
```

Initialisation avec un tableau

```
1  let myAnimals = new Set([' ', ' ', ' ', ' ']);
2
3  myAnimals.add([' ', ' ']);
4  myAnimals.add({ name: 'Rud', type: ' ' });
5  console.log(myAnimals.size); // 4
6
```

```
7 myAnimals.forEach(animal => {
8   console.log(animal);
9 });
10
11
12 // 
13 // 
14 // [" ", " "]
15 // Object { name: "Rud", type: " " }
```

Strings are a valid iterable so they can also be passed-in to initialize a set:

```
1 console.log('Only unique characters will be in this set.'.length); // 43
2
3 let sentence = new Set('Only unique characters will be in this set.');
```

```
4 console.log(sentence.size); // 18
```

On top of using *forEach* on a set, *for...of* loops can also be used to iterate over sets:

```
1 let moreAnimals = new Set([' ', ' ', ' ', ' ']);
2
3 for (let animal of moreAnimals) {
4   console.log(`Howdy ${ animal }`);
5 }
6
7 // Howdy 
8 // Howdy 
9 // Howdy 
10 // Howdy 
```

Keys and Values

Sets also have the *keys* and *values* methods, with **keys** being an alias for **values**, so both methods do exactly the same thing. Using either of these methods returns a new iterator

object with the values of the set in the same order in which they were added to the set.
Here's an example:

```
1 let partyItems = new Set([' ', ' ', ' ']);
2 let items = partyItems.values();
3
4 console.log(items.next());
5 console.log(items.next());
6 console.log(items.next());
7 console.log(items.next().done);
8
9 // Object {
10 //   done: false,
11 //   value: " "
12 // }
13
14 // Object {
15 //   done: false,
16 //   value: " "
17 // }
18
19 // Object {
20 //   done: false,
21 //   value: " "
22 // }
23
24 // true
```

Paramètres par défaut

```
1 let produit = function(nom = 'Sabre laser', prix = 220) {  
2   console.log(nom + " & " + prix);  
3 };  
4  
5 produit(undefined, 200); // Sabre laser & 200
```