Table of Contents

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 saisie par l'utilisateur et l'informer des erreurs et corrections a 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.

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.

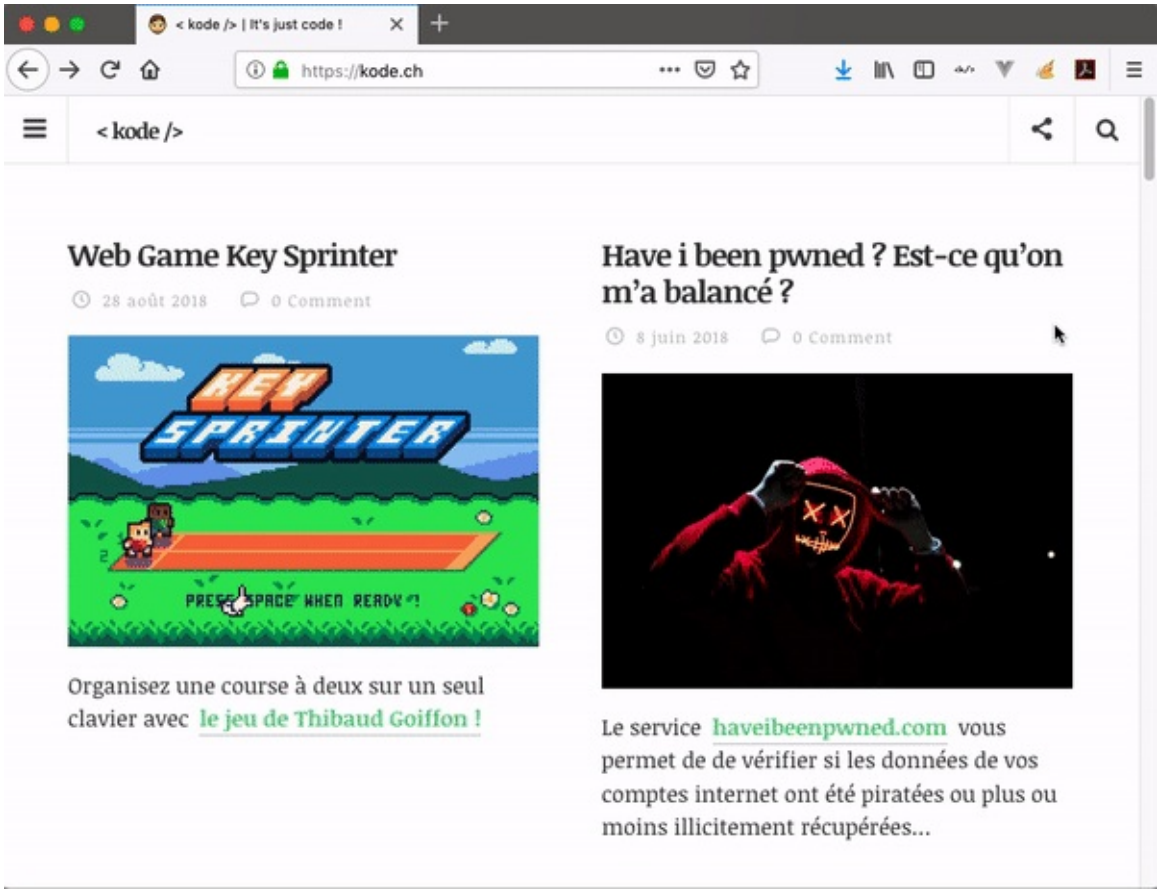
Quelques généralités sur JavaScript :

- 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



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.

```
<h1>Titre de ma page</h1>

<script>
    alert("Bonjour depuis une page HTML !");
</script>

<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>
```

Ou placer la balise `<script>`

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

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

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <title>Panier d'achats</title>
  </head>
  <body>
    <h1>Votre panier d'acahts</h1>
    <p>Cette semaine promotion sur les loutres blanches du Gabon</p>

    <!-- Inclusion des scipts -->
    <script src="panier-achats.js"></script>
  </body>
</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.

Commentaires & entêtes

```
// Ceci est une ligne de commentaires

/*
Ceci est un bloc
de commentaires
*/
```

Entête de document

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

Entête de fonction

```
/**
 * Additionne deux nombres.
 * @param {number} a - nombre a.
 * @param {number} b - nombre b.
 * @return {number} résultat de a + b.
 */
function add(a, b) {
    return a + b;
}
```

Entête de classe

```
/** Classe représentant un point. */
class Point {
    /**
     * Crée un point.
     * @param {number} x - coordonnée x.
     * @param {number} y - coordonnée y.
     */
    constructor(x, y) {
        // ...
    }
```

```
    /**
     * Retourne la coordonée x
     * @return {number} La coordonée x.
     */
    getX() {
        // ...
    }

    /**
     * Retourne la coordonée y
     * @return {number} La coordonée y.
     */
    getY() {
        // ...
    }

    /**
     * Converti en point une chaine contenant deux nombres séparés par un virgule.

     * @param {string} str - La chaine contenant les deux nombres séparés par une v
irgule.
     * @return {Point} Un objet Point.
     */
    static fromString(str) {
        // ...
    }
}
```

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` ≠ `nomdefamille`

Déclarer des variables et constantes

```
// Variables
var personneNom = "Dinateur";
let personneAge = 22;

// Constantes
const URL = "http://kode.ch";
const AGE_MAX = 65;
const langues = ['FR','EN'];
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.

``

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).

```
// Avec var
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // c'est la même variable !
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

// Avec let
function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // c'est une variable différente
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

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.

```
let nombre = 1;
let chaine = 'some Text';
let bools = true;
let tableau = [];
let objet = {};
let pasUnNombre = NaN; //NaN (Not A Number) est une valeur utilisée pour représente
r une quantité qui n'est pas un nombre
let vide = null;
let nonDefini;

typeof nombre;  // 'number'
typeof chaine;  // 'string'
typeof bools;   // 'boolean'
typeof tableau; // 'object' -- les tableaux sont de type objet.
```

```
typeof objet; // 'object'
typeof pasUnNombre; // 'number' -- Et oui NaN fait partie de l'objet Number.
typeof nonDefini; // 'undefined'
```

Exemple de test de type

```
let message = "Bonjour le monde";

if(typeof message === "string"){
    alert("c'est une chaine");
} else {
    alert("ce n'est pas une chaine !");
}
```

Astuces

Comment savoir si une variable contient un tableau ?

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

```
let tableau = ['je','suis','un','tableau'];

// Si est un objet et possède un propriété length
if(typeof tableau === 'object' && tableau.hasOwnProperty('length')) {
    alert("C'est un tableau !");
} else {
    alert("Ce n'est PAS un tableau !");
}
```

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.

```
let age = NaN;

// Si est de type number et est un nombre valide
if(typeof age === 'number' && !isNaN(age)) {
    alert("C'est un nombre !");
} else {
    alert("Ce n'est PAS un nombre !");
}
```

Conversions

Convertir des nombres en chaines de caractères

String()

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

```
let total = 123.56;
String(total); // "123.56"
String(123); // "123"
String(100 + 23); // "123"
```

.toString()

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

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

Opérateur + concaténation

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

```
let total = 123.56;
total + ""; // "123.56"
100 + "123"; // "100123"
100 + 23 + ""; //"123"
50 + " CHF"; // "50 CHF"
```

Convertir une chaîne de caractères en nombre

Il existe deux méthodes :

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

```
// Conversion nombre entier en base 10
parseInt("35", 10); // 35

// Conversion en base 2
```

```
parseInt("01010",2); // 10

// Conversion nombre entier (en base 10 si pas de deuxième paramètre)
parseInt("22 ans"); // 22

// Conversion nombre entier
parseInt("33.1045"); //33

// Conversion en nombre flottant
parseFloat("33.1045"); //33.1045

// Conversion en nombre flottant
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 `-=`.

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

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

Opérateur de concaténation de chaines

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 :

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

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

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 :

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

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

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

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

Conditions

`if... else`

```
// If Else
let a = 1;
let b = 2;

if (a < b) {
  console.log('Juste !');
} else {
  console.log('Faux !');
}


// Multi If Else
let a = 1;
let b = 2;
let c = 3;

if (a > b) {
  console.log('A plus grand que B');
} else if (a > c) {
  console.log('Mais A est plus grand que C');
} else {
  console.log('A est le plus petit');
}
```

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`

```
//Initialisation avec condition
let solde = 200;
let typeSolde = (solde < 0) ? "Négatif" : "Positif"; //"Positif"

//Si estMembre est vrai alors retourne 2$ sinon 10$
function prixEntree(estMembre) {
  return (estMembre ? "$2.00" : "$10.00");
}
```

Sélections avec `switch`

```
let fruit = 'Bananes';

switch (fruit) {
  case 'Oranges':
    console.log('Les oranges sont à 2.55€ le kilo');
    break;
  case 'Mangues':
  case 'Bananes':
    console.log('Les mangues et bananes sont à 7.70€ le kilo');
    break;
  default:
    console.log('Désolé, nous ne vendons pas de ' + fruit + ' !');
}

// Résultat : 'Les mangues et bananes sont à 7.70€ le kilo'
```

Boucles

While

```
let i = 0;
while (i < 4) {
  console.log(i);
  i += 1 // Eviter l'utilisation de "i++". Préférer "++i" ou "i += 1"
}

// 0
// 1
// 2
// 3
```

Do...while

```
let i = 0;
do {
  console.log(i);
  i += 1 // Eviter l'utilisation de "i++". Préférer "++i" ou "i += 1"
} while (i < 4)

// 0
// 1
// 2
// 3
```

For

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

// 0
// 1
// 2
// 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.

```
let animaux = [ '   ', '   ', '   ', '   '];

for (let animal of animaux) {
  console.log(animal);
}

//
//
//
//
```

For...in

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

```
//Création d'un objet personne
let personne = {
   nom: "Dinateur",
   prenom: "Laure",
   age: 33
};
//Parcours et affiche le nom et la valeur des propriétés de personne
for (let prop in personne) {
   console.log(prop + " => " + personne[prop]);
}

// nom => Dinateur
// prenom => Laure
// age => 33
```

Objets

Ajouter, modifier des propriétés

```
// Créer un nouvel objet
let personne = {};

// Ajouter un propriété
personne.prenom = 'Laure';
personne['nom'] = 'Dinateur'; //Autre syntaxe pour l'ajout

// Accéder à une propriété
personne.prenom; // Laure
personne.nom; // Dinateur

// Supprimer une propriété
delete personne.nom;
```

Tableaux

Ajouter et retirer des valeurs

```
// Crée un tableau vide
let monPremierTab = [];

// Crée un tableau avec valeurs. Peut contenir différents types
let monTab = [monPremierTab, 33, true, 'une chaine'];

// Retourne un élément spécifique du tableau
monTab[1]; // Retourne 33

// Changer une valeur
monTab[1] = "ok";

// Ajouter une valeur à la fin d'un tableau
monTab[monTab.length] = 'nouvelle valeur';

// Ajouter une ou plusieurs valeurs à la fin d'un tableau
monTab.push('fromage', 'pain');

//Ajouter une ou plusieurs valeur au début du tableau
monTab.unshift('poivre', 'sel');

// Récupérer et supprimer le dernier élément d'un tableau
let dernier = monTab.pop();

// Récupérer et supprimer le premier élément du tableau
let premier = monTab.shift();

// Récupérer et supprimer un sous tableau
// Premier paramètre postion de départ, 2e paramètre le nombre d'éléments
monTab.splice(3, 2); //Reourne et supprime le 4e et 5e élément
```

Parcourir un tableau

instruction for

```
let animaux = [ '   ', '   ', '   ', '   '];

// Boucle for classique (éviter i++ et utiliser ++i ou i+=1)
for (let i = 0; i < animaux.length; ++i) {
    console.log(animaux[i]);
}
```

```
//
//
//
//
```

instruction for...of

```
let animaux = ["   ", "   ", "   ", "   "];

// Itération avec for..of
for (let animal of animaux) {
   console.log(animal);
}

//
//
//
//
```

Méthode forEach()

```
let animaux = ["   ", "   ", "   ", "   "];

// Méthode forEach avec fontion anonyme (depsui ES5 seulement)
animaux.forEach(function(animal) {
   console.log(animal);
});

//
//
//
//
```

Exemples

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.

```
function bonjour() {
  alert('Bonjour !');
}

// Créer un timer et stoque son ID dans timerBonjour
// Le timer attendra 5000 millisecondes avant d'appeler la fonction bonjour()
let timerBonjour = window.setTimeout(bonjour, 5000);

// Annule le timer correspondant à l'ID passé en paramètre
window.clearTimeout(timerBonjour);
```

Exemple

```
<button onclick="startBonjour();">
  Affiche une alerte après 3 secondes...
</button>
<button onclick="stopBonjour();">
  Annuler l'affichage
</button>

<script>
let timerBonjour;

function bonjour() {
  alert("Bonjour !");
}

// Crée un timer qui appelle bonjour() après 3 secondes
// Stoque l'ID du timer dans la variable timerBonjour
function startBonjour() {
  timerBonjour = window.setTimeout(bonjour, 3000);
}

// Annule le timer timerBonjour
function stopBonjour() {
  window.clearTimeout(timerBonjour);
}
</script>
```

setInterval()

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

```
function bonjour() {
  alert('Bonjour !');
}

// Créer un intervalle et stoque son ID dans intervalleBonjour
// L'intervalle appellera bonjour() toutes les 5000 millisecondes
let intervalleBonjour = window.setInterval(bonjour, 5000);

// Annule l'intervalle correspondant à l'ID passé en paramètre
window.clearInterval(intervalleBonjour);
```

Exemple

```
<div>
   <p>pin-pon, pin-pon, pin-pon ...</p>
</div>

<button onclick="changeCouleur();">Start</button>
<button onclick="stopChangeCouleur();">Stop</button>

<script>
let intervalleCouleur;

// Crée un intervalle qui appelle flashText() toute les 500 millisecondes
function changeCouleur() {
   intervalleCouleur = setInterval(flashText, 500);
}

function flashText() {
   // Récupère 1er paragraphe du document
   let para = document.querySelector("p");

   // Change la couleur du texte en rouge ou en bleu
   if (para.style.color === "red") {
      para.style.color = "blue";
   } else {
      para.style.color = "red";
   }
}

// Annule l'intervalle
function stopChangeCouleur() {
   clearInterval(intervalleCouleur);
```

```
}
</script>
```

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 pricipaux type de nœud :

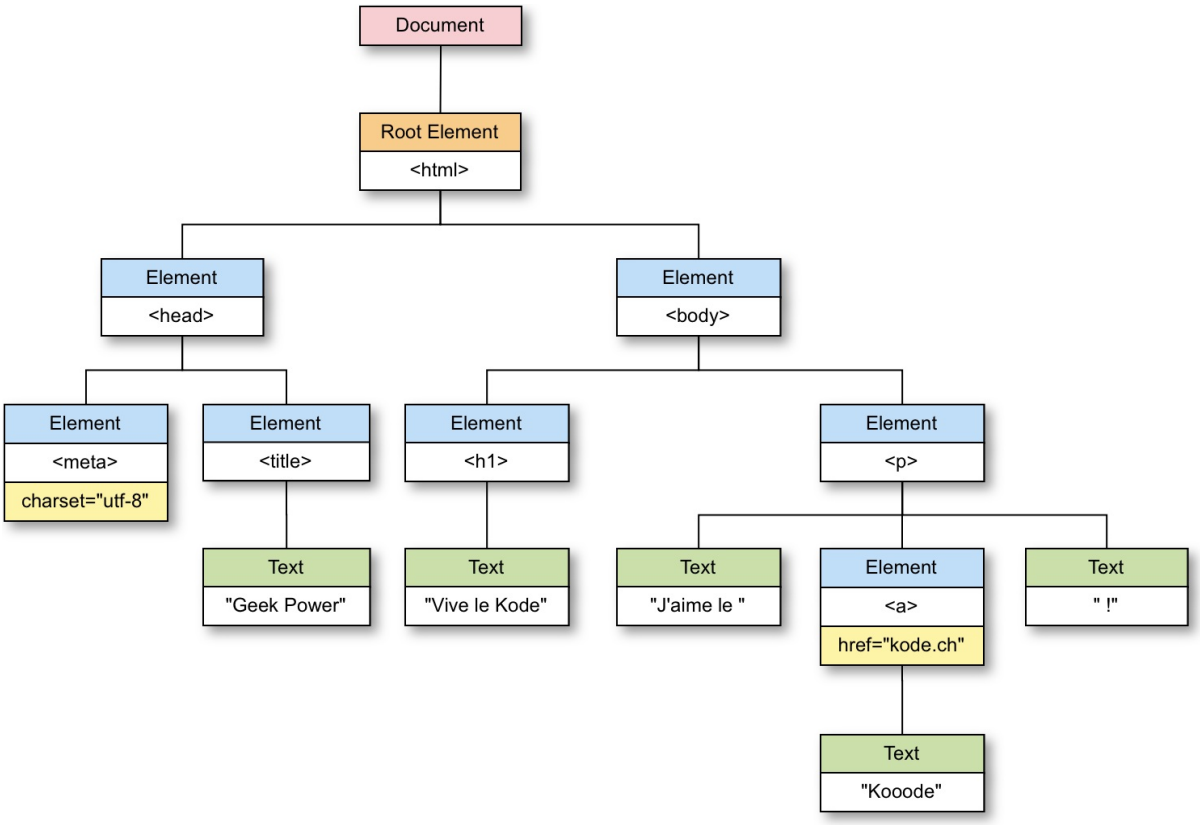
- `Element` : un élément HTML `<p>, <h1>, <body>, <img>, ...`
- `Text` : chaine de caractères `"C'est pas faux !"`
- `Comment` : commenaire HTML `<!-- Je suis un simple commentaire -->`

Exemple

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

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Geek Power</title>
    </head>

    <body>
        <h1>Vive le Kode</h1>
        <p>J'aime le <a href="http://www.kode.ch">Kooode</a> !</p>
    </body>
</html>
```



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 élements fils, parents ou voisins (frères)
- Modifier un élément en changeant :
  - son contenu texte `"texte"` ou HTML `"<li>Yoda</li>"`
  - 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.

```
// Renvoie l'élément avec l'id "menu" <nav id="menu">...</nav>
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.

```
// Renvoie un tableau de tous les éléments du document
// appartenant à la classe rouge
let elementsRouges = document.getElementsByClassName('rouge');

// Renvoie un tableau de tous les enfants de l'élément spécifié
// appartenant aux classes rouge ET gras
let elementsRougesGras = monElement.getElementsByClassName('rouge gras');
```

Element.getElementsByTagName()

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.

```
// Renvoie un tableau de tous les éléments <li> du document
let elementsDeListes = document.getElementsByTagName('li');

// Renvoie un tableau des éléments <strong> enfants de monElement
let 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.

```
// Revoie le premier paragraphe du document
const PREMIER_PARA = document.querySelector('p');

// Renvoie le premier élément du document correspondant à l'un des sélecteur CSS
// 'img.rouge, img-jaune' (images appartenant à la classe rouge OU jaune)
const IMG_ROUGE_OU_JAUNE = document.querySelector('img.rouge, img.jaune');

// Renvoie la valeur de l'élément coché (:checked) du groupe d'input "pays"
let pays = document.querySelector('input[name="pays"]:checked').value;

// Renvoi le champ texte "login" contenu dans une div avec la classe ".utilisateur"
const INPUT_LOGIN = 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 correspondent au sélecteur CSS - ou groupe de sélecteurs - spécifié(s).

```
// Retourne tous les paragraphes du document
let paras = document.querySelectorAll("p");

// Retourne tous les paragraphes présents dans une div avec la classe "article"
let parasAticle = container.querySelectorAll("div.article > p");

// Retourne un tableau de tous les éléments correspondants à l'un des sélecteurs
// Sélectionne les div appartenant à la classe "note" OU "alert"
let 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.

```
<p id="intro">
  Je suis un <strong>joli</strong> paragraphe !
</p>

<script>
// Récupère le paragraphe #intro
let introPara = document.getElementById('intro');
// Récupère le contenu HTML du paragraphe
let contenu = introPara.innerHTML; //Je suis un <strong>joli</strong> paragraphe !

// Remplacer le contenu HTML du paragraphe
introPara.innerHTML = 'Je suis un <em>nouveau</em> paragraphe !';
// <p>Je suis un <em>nouveau</em> paragraphe !</p>

// Ajouter du contenu HTML à la fin du contenu existant avec +=
introPara.innerHTML += ' <a href="http://kode.ch">Lien</a>';
// <p>Je suis un <em>nouveau</em> paragraphe ! <a href="http://kode.ch">Lien</a></p
>
</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 sélectionnnait le contenu d'un élément avec le curseur, et le copiait dans le presse-papier.

```
<p id="intro">
  Je suis un <strong>joli</strong> paragraphe !
</p>
```

```
<script>
// Récupère le paragraphe #intro
let introPara = document.getElementById('intro');

// Récupération du contenu avec innerText
console.log(introPara.innerText); // Je suis un joli paragraphe !

// Récupération du contenu avec innerHTML
console.log(introPara.innerHTML); // Je suis un <strong>joli</strong> paragraphe !
</script>
```

`insertAdjacentHTML(position, text);`

Permet d'ajouter du `text` HTML à une `position` donnée autours 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.

```
<p>bla bla</p>
<!-- beforebegin -->
<p id='intro'>
  <!-- afterbegin -->
  Un peu de texte
  <!-- beforeend -->
</p>
<!-- afterend -->
<p>bla bla</p>
```

Exemple

```
<ul id="liste1">
  <li>élément de #liste1</li>
</ul>

<ul id="liste2">
  <li>élément de #liste2</li>
</ul>

<script>
// Ajouter un nouveau contenu entre #liste1 et #liste2
let liste2 = document.getElementById('liste2');
```

```
// Ajout de <p> juste avant et juste après #liste2
liste2.insertAdjacentHTML('beforebegin', '<p>Juste avant #liste2</p>');
liste2.insertAdjacentHTML('afterend', '<p>Juste après #liste2</p>');

// Ajout de <li> comme premier et dernier fils de la #liste2
liste2.insertAdjacentHTML('afterbegin', '<li>Nouveau premier fils de #liste2</li>')
;
liste2.insertAdjacentHTML('beforeend', '<li>Nouveau dernier fils de #liste2</li>');
</script>
```

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 chaines 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`.

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

```
let intros = document.getElementsByClassName("intro");

for (let i = 0; i < intros.length; i = i + 1) {
    intros[i].style.fontSize = '1.5em';
```

```
    intros[i].style.backgroundColor = 'lime';
}
```

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

```
// Récupère #intro
const INTRO = document.getElementById('intro');
// Récupère le style CSS de #intro
let styleINTRO = window.getComputedStyle(INTRO);
// Affiche la valeur de la propriété CSS top de #intro
console.log( styleINTRO.getPropertyValue('top') );
```

Modifier les attributs

```
let colmar = document.getElementById("colmar");
colmar.href   = 'http://www.colmar.fr';
colmar.target = '_blank';

// Avec la méthode setAttribute
colmar.setAttribute('href', 'http://www.colmar.fr');
colmar.setAttribute('target', '_blank');
```

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

Modifier les classes CSS

```
// Récupère l'élément #menu
let menu = document.getElementById('menu');

// Supprime la class rouge de #menu si présente
menu.classList.remove('rouge');

// Ajoute la class vert à #menu si non présente
menu.classList.add('vert');

// Ajoute ou retire plusieurs classes
menu.classList.add('jaune', 'bleu');
menu.classList.remove('jaune', 'bleu');

/* Alternance :
    Si #menu a la classe .rouge toggle('rouge') la retire
    Si #menu n'a pas la classe .rouge toggle('rouge') l'ajoute */
menu.classList.toggle('rouge');

// Retoune true si #menu a la classe .rouge, false s'il ne l'a pas
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("chaine 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 `<li>2kg de Pain</li>` à la fin de la liste `#fondue` .

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

<script>
// 1. Création du nouvel élément <li>
let newLi = document.createElement('li');

// 2. Création du nœud texte
let newLiTexte = document.createTextNode("2kg de Pain");

// 3. Ajout du texte au <li>
newLi.appendChild(newLiTexte);

// 4. Récupération de la liste
let listeFondue = document.getElementById('fondue');

// 5. Ajoute le nouvel élément <li> à la fin de la liste
listeFondue.appendChild(newLi);
</script>
```

Résultat

```
<ul id="fondue">
    <li>2kg de Fromage</li>
    <li>1L de Kirsh</li>
    <li>2kg de Pain</li>
</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 un 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 `<li>2kg de Pain</li>` au début de la liste `#fondue` .

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

<script>
// 1. Création du nouvel élément <li>
let newLi = document.createElement('li');

// 2. Création du nœud texte
let newLiTexte = document.createTextNode("2kg de Pain");

// 3. Ajout du texte au <li>
newLi.appendChild(newLiTexte);

// 4. Récupération d'un 1er élément <li> de la liste actuelle existant du DOM
let premierLi = document.querySelector('#fondue li:first-child');

// 3. Récupération du parent de premierLi
let parentLi = premierLi.parentNode;

// 6. Ajout de newLi avant premierLi
parentLi.insertBefore(newLi, premierLi);
</script>
```

Résultat

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

```
</ul>
```

Supprimer, remplacer et cloner

Supprimer un élément

```
<div>
    <h1>Un Titre</h1>
    <p>Petit paragraphe<p>
</div>

<script>
// Récupération du 1er paragraphe de la 1re div du document
const P1 = document.querySelector("div p");
// Suppression du 1er paragraphe
P1.remove();
</script>
```

Supprimer un élément fils

```
<div>
    <h1>Un Titre</h1>
    <p>Petit paragraphe<p>
</div>

<script>
// Récupération de la 1re div du document
const DIV = document.querySelector("div");
// Récupération du 1er <p> de la DIV
const P1 = DIV.querySelector("p");
// Suppression du 1er paragraphe
DIV.removeChild(P1);
</script>
```

Remplacer un élément

```
<div>
   <h1>Un Titre</h1>
   <p>Petit paragraphe<p>
</div>

<script>
// Récupération de la 1re div du document
const DIV = document.querySelector("div");
// Récupération du 1er <p> de la DIV
const ANCIEN_PARA = DIV.querySelector("p");
```

```
// Création d'un nouveau <p>
const NOUVEAU_PARA = document.createElement("p");
// Modification du texte du nouveau <p>
NOUVEAU_PARA.innerText = 'Nouveau paragraphe';
// Remplace l'ancien <p> par le nouveau
DIV.replaceChild(NOUVEAU_PARA, ANCIEN_PARA);
</script>
```

Cloner un élément

```
<ul id="liste1">
   <li>Fromage</li>
   <li>Thé</li>
</ul>

<ul id="liste2">
   <li>Eau</li>
   <li>Sucre</li>
</ul>

<script>
// Récupération du dernier fils de #liste1 <li>Thé</li>
const DERNIER_FILS_LISTE1 = document.querySelector("#liste1 :last-child");
// Clone, copie, le dernier fils et son contenu, sa descendance
const CLONE = DERNIER_FILS_LISTE1.cloneNode(true);
// Ajoute le clone à la fin de #liste1
document.getElementById("liste2").appendChild(CLONE);
</script>
```

Evénements

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

Evénement DOM	Description
`click`	Bouton de la souris enfoncé puis relâché sur un élément.
`dblclick`	Deux foix l'événement `click`
`mouseover`	Souris au-dessus d'un élément.
`mouseout`	Souris sort d'un élément.
`mousedown`	Bouton de la souris enfoncé, pas relâché, sur un élément.
`mouseup`	Bouton de la souris relâché sur un élément.
`mousemove`	Souris en mouvement au-dessus d'un élément.
`keydown`	Touche clavier enfoncée, pas relâchée, sur un élément.
`keyup`	Touche clavier relâchée sur un élément.
`keypress`	Touche clavier enfoncée et relâchée sur un élément.
`focus`	L'élément reçoit, gagne, le focus. Quand un objet devient l'élément actif du document.
`blur`	Elément perd le focus.
`change`	Changement de a valeur d'un élément de formulaire.
`select`	Sélection du texte d'un élémen, mis en srubrillance.
`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 écouteur d'événement (listener) avec la méthode `addEventListener()`

> Le meilleure 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 nommées selon l'événement lié : `onclick`, `onkeypress`, `onfocus`, `onsubmit`, etc.

> Liste des gestionnaires d'événements : https://www.w3schools.com/tags/ref_eventattributes.asp

On peut scpécifier 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()`.

```
function citationLeia() {
  alert("Plutôt embrasser un Wookie");
}

// Affecte la fonction citationLeia() au click du bouton
document.querySelector('button').onclick = citationLeia;

// Variante avec fonction anonyme
document.querySelector('button').onclick = function() {
  alert("Plutôt embrasser un Wookie");
};

// Variante avec fonction fléchée (arrow function)
document.querySelector('button').onclick = () => alert("Plutôt embrasser un Wookie"
);
```

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);
```

```
let newElement = document.getElementsByTagName('h1');

newElement.onclick = function() {
  console.log('clicked');
};

let logEventType = function(e) {
    console.log('event type:', e.type);
};

newElement.addEventListener('focus', logEventType, false);
newElement.removeEventListener('focus', logEventType, false);

window.onload = function() {
  console.log('Im loaded');
};
```

Ajouter un événement à une liste d'éléments

```
let boutons = document.querySelectorAll("button");

for (let bouton of boutons) {
   bouton.addEventListener("click", function(event) {
      bouton.classList.toggle("rouge");
   });
}
```

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

```
button {
   cursor: pointer;
   color: #7f8c8d;
   font-weight: bold;
   background-color: #ecf0f1;
   padding: 1em 2em;
   border: 2px solid #7f8c8d;
}

button.rouge {
   border-color: #c0392b;
   background-color: #e74c3c;
```

```
    color: #ecf0f1;
}
```

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` .

```
<button>Clique moi !</button>

<script>
// Récupère le 1er boutons du document
const BOUTON = document.querySelector("button");
// Ajoute événement click avec une fonction avec paramètre event
BOUTON.addEventListener("click", function (event) {
  // Affiche le type d'événement envoyé
  alert(event.type); // click
});
</script>
```

Récupérer la cible d'un événement

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

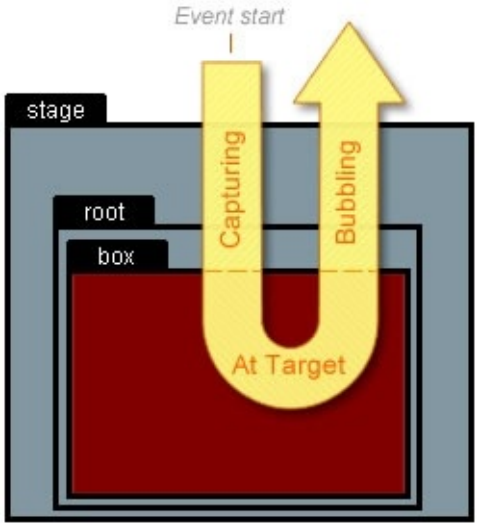
```
<button>Clique moi !</button>

<script>
// Récupère le 1er boutons du document
const BOUTON = document.querySelector("button");
// Ajoute événement click avec une fonction avec paramètre event
BOUTON.addEventListener("click", function (event) {
  // Récupère l'élément qui a envoyé l'événement, la cible
  let cible = event.target;
  // Modifie la taille du texte de la cible
  cible.style.fontSize = "2em";
  // Affiche le contenu texte de la cible
  alert(cible.innerText); // Clique moi !
});
</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.




```
<div id="div1">
  <p id="p1">I am Bubbling</p>
</div><br>

<div id="div2">
  <p id="p2">I am Capturing.</p>
</div>

<script>
document.getElementById("p1").addEventListener("click", function() {
    alert("You clicked the P element!");
}, false);

document.getElementById("div1").addEventListener("click", function() {
    alert("You clicked the DIV element!");
}, false);

document.getElementById("p2").addEventListener("click", function() {
    alert("You clicked the P element!");
}, true);

document.getElementById("div2").addEventListener("click", function() {
    alert("You clicked the DIV element!");
}, true);
</script>
```

A lire...

Naviguer dans le DOM

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

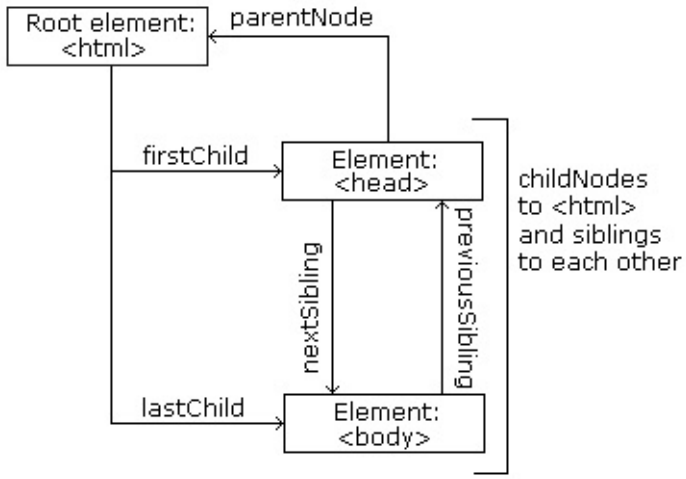


Tableau des propriétés

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

```
<div>Bonjour le <strong>monde</strong>!</div>

<script>
// Récupère la première <div> du document
let div = document.querySelector("div");

// Affiche le premier fils avec firstChild
console.log( div.firstChild ); // "Bonjour le"

// Affiche le premier fils de type Element avec firstElementChild
console.log( div.firstElementChild ); // "<strong>monde</strong>"
</script>
```

Propriétés de type `Node`

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

<script>
// Récupère le premier élément strong du document
let strongElement = document.querySelector("strong");

// Noeud parent
moneElement.parentNode; // <div>

// Tous les fils (tableau de noeuds)
monElement.childNodes; // ["le ", <em>]

// Premier fils
monElement.firstChild; // "le "

// Dernier fils
monElement.lastChild; // <em>

// Frêre suivant
monElement.nextSibling; // "!"

// Frêre précédent
monElement.previousSibling; // "Bonjour "
</script>
```

Propriétés de type `Element`

```
<div id="animaux">
   <h1>Animaux</h1>
   <ul>
      <li>  </li>
      <li>  </li>
   </ul>
</div>

<script>
// Récupère la première liste non triée du document
let listeAnimaux = document.querySelector("ul");

// Noeud parent
listeAnimaux.parentNode; // <div id="animaux">...</div>

// Tous les fils (tableau de noeuds)
listeAnimaux.children; // [<li>  </li>, <li>  </li>]

// Premier fils
listeAnimaux.firstElementChild; // <li>  </li>

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

Formulaires

Envoyer des formulaires

Envoyer et réinitialiser un formulaire

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

```
// Récupère le 1er formulaire du document
const FORMULAIRE = document.querySelector('form');

// Envoyer un formulaire
FORMULAIRE.submit();

// Réinitialiser un formulaire
FORMULAIRE.reset();
```

Événement `submit`

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

```
const FORMULAIRE = document.querySelector('form');

FORMULAIRE.addEventListener('submit', function(){
   console.log("Formulaire envoyé !");
});
```

Une foix 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.

```
const FORMULAIRE = document.querySelector('form');

// Ne pas oublier d'ajouter un paramètre à la fonction pour récupérer l'événment.
FORMULAIRE.addEventListener('submit', function(event){
   event.preventDefault(); // Stoppe l'envoi du formulaire
   console.log("Formulaire envoyé !");
});
```

Exemple

```
<form action="https://kode.ch/getpost/" method="post">
   <label for="nom">Votre nom</label>
   <input type="text" id="nom" name="nom">
   <button>Envoyer</button>
</form>

<script>
// 1er formulaire du document
const FORMULAIRE = document.querySelector('form');
// Champ texte nom
const TXT_NOM = document.getElementById('nom');

// Evénement submit => Lors de l'envoi du formulaire
FORMULAIRE.addEventListener('submit', function(event){
   // Désactive l'envoi du formulaire
   event.preventDefault();

   // Si utilisateur n'a pas saisi de nom
   if(TXT_NOM.value === "") {
      alert("Entrez votre nom !");
      return; // Sors de la fonction
   }

   // Envoie le formulaire
   FORMULAIRE.submit();
});
</script>
```

Récupérer la valeur des champs

Champs de saisie

Propriété `value`

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

```
monElement.value;
```

Liste déroulantes

Propriété `value`

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

```
monElementSelect.value;
```

Evénement `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

```
<select name="pays" id="pays">
   <option selected value="">-- Sélectionnez un pays --</option>
   <option value="FR">France</option>
   <option value="IT">Italie</option>
   <option value="CH">Suisse</option>
</select>

<div>
   Code du pays : <span class="code"></span>
</div>

<script>
// Récupère la liste déroulante #pays et le span .code
const LIS_PAYS = document.getElementById("pays");
```

```
const SPAN_CODE = document.querySelector("span.code");

// Sur changement de la valeur de la liste déroulante
LIS_PAYS.addEventListener("change", function() {
   // Récupère la valeur de l'option sélectionnée
   let codePays = LIS_PAYS.value;
   // Modifie le contenu texte du span .code
   SPAN_CODE.innerText = codePays;
});
</script>
```

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

```
<form action="https://kode.ch/getpost/" method="post">
   <input type="checkbox" id="copie" name="copie">
   <label for="copie">Recevoir une copie</label>
   <button>Envoyer</button>
</form>

<script>
// 1er formulaire du document
const FORMULAIRE = document.querySelector('form');
// Case à cocher "copie"
const CHK_COPIE = document.getElementById('copie');

// Evénement submit => Lors de l'envoi du formulaire
FORMULAIRE.addEventListener('submit', function(event){
   // Si utilisateur n'a pas saisi de nom
   if(CHK_COPIE.checked === true) {
      alert("Message envoyé AVEC copie !");
   } else {
      alert("Message envoyé SANS copie !");
   }
});
</script>
```

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"]` .

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

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

```
<form action="https://kode.ch/getpost/" method="post">
   <input type="checkbox" name="couleurs[]" id="rouge" value="rouge">
   <label for="rouge">Rouge</label>

   <input type="checkbox" name="couleurs[]" id="vert" value="vert">
   <label for="vert">Vert</label>

   <input type="checkbox" name="couleurs[]" id="bleu" value="bleu">
   <label for="bleu">Bleu</label>

   <button>Envoyer</button>
</form>

<script>
// 1er formulaire du document
const FORMULAIRE = document.querySelector("form");

// Evénement submit => Lors de l'envoi du formulaire
FORMULAIRE.addEventListener("submit", function(event) {
   event.preventDefault();

   // Cases cochée dans le groupe couleurs[]
   let couleursCochées = document.querySelectorAll(
      'input[name="couleurs[]"]:checked'
   );

   //Récupère la valeur des éléments cochés
   for (let couleur of couleursCochées) {
      alert(couleur.value);
   }
});
```

```
</script>
```

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"]` .

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

Exemple

```
<form action="https://kode.ch/getpost/" method="post">
   <input type="radio" name="genre" id="h" value="Homme">
   <label for="h">Homme</label>

   <input type="radio" name="genre" id="f" value="Femme">
   <label for="f">Femme</label>

   <button>Envoyer</button>
</form>

<script>
// 1er formulaire du document
const FORMULAIRE = document.querySelector("form");

// Evénement submit => Lors de l'envoi du formulaire
FORMULAIRE.addEventListener("submit", function(event) {
    // Désactive l'envoi du formulaire
   event.preventDefault();

   // Radio coché dans le groupe genre
   let genre = document.querySelector(
      '[name="genre"]:checked'
   );

   // Test si un genre est coché
   if(genre === null) {
      alert("Sélectionner un genre !");
      return;
   }

   alert(genre.value);
});
</script>
```

Valider les saisies utilisateurs

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

```
<ul class="message"></ul>
<form action="https://kode.ch/getpost/" method="post">
   <ul>
      <li>
         <label for="nom">Votre nom</label>
         <input type="text" id="nom" name="nom">
      </li>
      <li>
         <label for="age">Votre age</label>
         <input type="text" id="age" name="age">
      </li>
      <li>
         <button type="submit">Envoyer</button>
      </li>
   </ul>
</form>
```

```
/**
 * Valide le nom et l'âge d'une personne et retourne un tableau d'erreurs
 * @return {Array} Tableau de messages d'erreur
 */
function validerPersonne(nom, age) {
   // Initialisation du tableau des erreurs
   let erreurs = [];

   //Supprime les espaces en début et fin de chaine
   nom = nom.trim();

   //Converti l'age en entier
   age = parseInt(age);

   // Si le nom vide
   if (nom === "") {
      erreurs.push("Entrez un nom !");
   }

   // Si l'âge n'est pas un nombre entier compris entre 0 et 120
   if (Number.isNaN(age) || age < 1 || age > 119) {
      erreurs.push("Entrez un age valide !");
   }

   return erreurs;
}
```

```
/**
 * Ajoute le contenu d'un tableau à la fin d'une liste HTML
 * @param {HTMLElement} eleListe - Liste HTML (ol ou ul) à remplir
 * @param {Array} erreurs - tableau de String
 */
function ajouterFinListe(eleListe, erreurs) {
   // Parcours les messages d'erreur
   for (message of erreurs) {
      // Ajoute un li au contenu de la liste
      eleListe.innerHTML += "<li>" + message.toString() + "</li>";
   }
}

// Récupération du formulaire et de la liste message
const eleFormulaire = document.querySelector("form");
const eleMessage = document.querySelector("ul.message");

// Evénement submit => Lors de l'envoi du formulaire
eleFormulaire.addEventListener("submit", function(event) {
   // Désactive l'envoi du formulaire
   event.preventDefault();

   // Récupère les champs nom et age
   const txtNom = document.getElementById("nom");
   const txtAge = document.getElementById("age");

   // Supprime les anciens messages d'erreur
   eleMessage.innerHTML = "";

   // Validation des données
   let erreurs = validerPersonne(txtNom.value, txtAge.value);

   // Si il y a des erreurs
   if (erreurs.length > 0) {
      // Ajoute les erreurs à la fin de ul.message
      ajouterFinListe(eleMessage, erreurs);
   } else {
      // Envoi du formulaire
      eleFormulaire.submit();
   }
});
```

```
body {
   font-family: "Trebuchet MS", Helvetica, sans-serif;
}

ul.message {
   color: #ecf0f1;
   background-color: #e74c3c;
```

```
}

ul.message li {
   padding: .5em 0;
}

form ul {
   list-style-type: none;
   padding: 0;
}

form ul li {
   padding: 0 0 1em 0;
}

form ul li label {
   display: block;
   font-weight: bold;
}
```

JavaScript Moderne

Transpiller

Template Literals

Les Template literals permettent d'écrire des chaines de caractètes 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}`

```
let nom = 'Skywlaker';
let prenom = 'Luc'
let message = `Salut ${prenom} ${nom} !`;

alert(message); // Salut Luc Skywlaker !

let ficheClient = `<div class="fiche-client">
                        <div>Nom : ${nom}</div>
                           <div>Prenom : ${prenom}</div>
                           <button>Appeler papa</button>
                    </div>`;

document.write(ficheClient);
```

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`.

```
let animals = new Set();

animals.add('  ');
animals.add('  ');
animals.add('  ');
animals.add('  ');
console.log(animals.size); // 4
animals.add('  ');
console.log(animals.size); // 4

console.log(animals.has('  ')); // true
animals.delete('  ');
console.log(animals.has('  ')); // false

animals.forEach(animal => {
  console.log(`Hey ${animal}!`);
});

// Hey   !
// Hey   !
// Hey   !

animals.clear();
console.log(animals.size); // 0
```

Initialisation avec un tableau

```
let myAnimals = new Set(['  ', '  ', '  ', '  ']);

myAnimals.add(['  ', '  ']);
myAnimals.add({ name: 'Rud', type: '  ' });
console.log(myAnimals.size); // 4

myAnimals.forEach(animal => {
  console.log(animal);
});


//
```

```
//
// ["  ", "  "]
// Object { name: "Rud", type: "  " }
```

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

```
console.log('Only unique characters will be in this set.'.length); // 43

let sentence = new Set('Only unique characters will be in this set.');
console.log(sentence.size); // 18
```

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

```
let moreAnimals = new Set(['  ', '  ', '  ', '  ']);

for (let animal of moreAnimals) {
  console.log(`Howdy ${ animal }`);
}

// Howdy
// Howdy
// Howdy
// 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:

```
let partyItems = new Set(['  ', '  ', '  ']);
let items = partyItems.values();

console.log(items.next());
console.log(items.next());
console.log(items.next());
console.log(items.next().done);

// Object {
//   done: false,
//   value: "  "
// }

// Object {
//   done: false,
//   value: "  "
// }
```

```
// Object {
//   done: false,
//   value: "  "
// }

// true
```

Paramètres par défaut

```
let produit = function(nom = 'Sabre laser', prix = 220) {
    console.log(nom + " & " + prix);
};

produit(undefined, 200); // Sabre laser & 200
```