**Javascript suite**

**Les tableaux**

**1. Tableaux, introduction**

Un tableau (appelé array en anglais) est un type avancé de valeurs. C’est une liste ordonnée de valeurs.

Exemple :

let fruits = [ 'Mangue', 'Raisin', 'Figue' ];

**Caractéristiques**

Comme tous les autres types de valeur, un tableau peut être stocké dans une variable.

En revanche, en JavaScript, les tableaux sont considérés comme une forme particulière du type object. Autrement dit, typeof [] retourne "object".

Comme toute variable en JavaScript, les valeurs stockées dans un tableau peuvent être de n’importe quel type (y compris un tableau), et n’ont pas besoin d’être tous du même type :

let fourreTout = [ null, true, 'bonjour', 1.2, fruits, undefined ];

Tout tableau possède une propriété length qui vaut le nombre de valeurs qu’il contient:

fourreTout.length; // => retourne 6, car il y a 6 valeurs dans notre tableau

**Accéder aux éléments d’un tableau**

Chaque élément d’un tableau est numéroté par ce qu’on appelle un “indice” (ou “index” en anglais).

A noter que cette numérotation commence par 0 (zéro), et non par 1.

Du coup, l’indice du premier élément est 0, et l’indice du dernier élément est length - 1 (length étant le nombre d’éléments du tableau).

On peut accéder à un élément de tableau en précisant son indice entre crochets :

let fruits = [ 'Mangue', 'Raisin', 'Figue' ];

fruits[0]; // vaut 'Mangue'

fruits[1]; // vaut 'Raisin'

fruits[2]; // vaut 'Figue'

fruits[3]; // undefined

**Modification d’un élément**

Pour modifier la valeur d’un élément, il suffit de l’adresser par indice (comme vu juste avant) puis de lui affecter une valeur, comme on le ferait pour une variable :

let fruits = [ 'Mangue', 'Raisin', 'Figue' ];

fruits[1] = 'Kiwi';

fruits; // => vaut [ 'Mangue', 'Kiwi', 'Figue' ]

**Ajout d’élément**

La méthode push() permet d’ajouter un élément à la fin du tableau. La valeur de cet élément est à passer en paramètre (entre parenthèses) de la méthode, quand on l’appelle:

let fruits = [ 'Mangue', 'Raisin', 'Figue' ];

fruits.push('Banane'); // appel de la méthode push() sur le tableau fruits, avec `Banane` en paramètre

fruits; // => [ 'Mangue', 'Raisin', 'Figue', 'Banane' ]

À chaque fois qu’on ajoute un élément à un tableau en appelant la méthode push(), sa longueur length est incrémentée (c.a.d. sa valeur augmente de 1):

fruits.length; // => 4, désormais

**Retrait d’élément**

La méthode pop() retourne la dernière valeur du tableau puis la retire de ce tableau:

let fruits = [ 'Mangue', 'Raisin', 'Figue' ];

let f = fruits.pop();

console.log(f); // => `Figue`

fruits; // => [ 'Mangue', 'Raisin' ];

À chaque fois qu’on retire un élément d’un tableau en appelant la méthode pop(), sa longueur length est décrémentée (c.a.d. sa valeur décroit de 1):

fruits.length; // => 2, désormais

**Exercice 7 Test calendrier**

**2. Tableaux, fonctions avancées**

**Recherche d’élément par valeur**

La méthode indexOf() retourne l’indice du premier élément d’un tableau, pour une valeur donnée. La valeur recherchée est à passer en paramètre (entre parenthèses) de la méthode, quand on l’appelle :

var fruits = [ 'Mangue', 'Raisin', 'Figue', 'Raisin' ];

fruits.indexOf('Raisin'); // => 1

Si aucun élément du tableau ne contient la valeur cherchée, l’appel à indexOf() retourne -1:

fruits.indexOf('Pomme'); // => -1, car il n'y a pas de valeur 'Pomme' dans fruits

**Concaténation de tableaux**

Comme pour les chaînes de caractères, il est possible de concaténer des tableaux.

La concaténation de deux tableaux consiste à créer un nouveau tableau contenant les éléments de ces deux tableaux.

La méthode concat() retourne un nouveau tableau contenant les éléments du tableau sur lequel elle est appelée, et d’un autre tableau passé en paramètre:

var fruits1 = [ 'Mangue', 'Raisin' ];

var fruits2 = [ 'Figue', 'Kiwi' ];

fruits1.concat(fruits2); // => [ 'Mangue', 'Raisin', 'Figue', 'Kiwi' ]

Il est bien entendu possible d’utiliser des tableaux littéraux, au lieu de variables :

[ 'Mangue', 'Raisin' ].concat([ 'Figue', 'Kiwi' ]); // => [ 'Mangue', 'Raisin', 'Figue', 'Kiwi' ]

**Partitionnement de tableaux**

La méthode slice() retourne un nouveau tableau contenant un extrait du tableau sur lequel elle est appelée.

A chaque appel de cette méthode, il faut fournir deux paramètres : l’indice de l’élément où commence cet extrait, et l’indice de l’élément où se termine l’extrait (non compris):

var fruits = [ 'Mangue', 'Raisin', 'Figue', 'Kiwi' ];

fruits.slice(1, 3); // => [ 'Raisin', 'Figue' ]

**Altération de tableau**

La méthode splice() (à ne pas confondre avec slice()) permet à la fois de supprimer et d’insérer des éléments dans un tableau, en fournissant leur indice(s).

Syntaxe d’appel de la méthode : tableau.splice(i, c, v1, v2, ...), avec

* paramètre i: indice à partir duquel on va effectuer la modification
* paramètre c: nombre d’éléments à supprimer depuis l’indice i
* paramètre(s) suivant(s): valeur(s) d’élément(s) à insérer à partir de l’indice i

Voici un exemple utilisant les paramètres i et c: (suppression seulement)

var fruits = [ 'Mangue', 'Raisin', 'Figue', 'Kiwi' ];

fruits.splice(1, 2); // depuis l'indice 1, supprimer 2 valeurs

fruits; // => [ 'Mangue', 'Kiwi' ]

Voici un exemple utilisant tous les paramètres : (insertion seulement)

var fruits = [ 'Mangue', 'Raisin', 'Figue', 'Kiwi' ];

fruits.splice(1, 0, 'Pomme'); // depuis l'indice 1, supprimer 0 valeurs, puis y insérer 'Pomme'

fruits; // => [ 'Mangue', 'Pomme', 'Raisin', 'Figue', 'Kiwi' ]

Contrairement aux méthodes concat() et slice(), splice() ne retourne pas un nouveau tableau, mais modifie le tableau sur lequel elle est appelée.

**Exercice Epicerie et Jeu du Pendu**

**Les objets**

**Objets JavaScript**

Dans la première moitié de ce chapitre nous avons vu que les tableaux permettaient de stocker une liste ordonnée d’éléments (ayant chacun une valeur de n’importe quel type). Et que ces éléments étaient indexés par des numéros.

Exemple:

var tableau = ['a', 'b', 4];

// => Correspondance entre numéros d'éléments et leur valeur

// 0: 'a'

// 1: 'b'

// 2: 4

Les objets permettent aussi de stocker des valeurs.

Alors que chaque valeur d’un tableau est stockée dans un **élément**, chaque valeur d’un objet est stockée dans ce qu’on appelle une **propriété**.

Alors que chaque élément d’un tableau est indexé par un numéro (indice), chaque propriété d’un objet est indexée par une chaîne de caractères.

**Définition d’objet**

Contrairement à la manière de définir un tableau, chaque valeur de propriété stockée dans un objet doit être précédée du nom de cette propriété (aussi appelé **clé**, ou **key** en anglais), et suivi par : (deux points).

let objet = {

1: 'Imane',

prop2: 'b',

prop3: 4,

};

// => Correspondance entre noms de propriétés et leur valeur

// 'prop1': 'a'

// 'prop2': 'b'

// 'prop3': 4

À noter que chaque définition de propriété doit être ponctuée d’une virgule.

Les règles d’indentation s’appliquent aussi à la définition d’objets sur plusieurs lignes, car les propriétés sont définies entre accolades.

Le format JSON (JavaScript Object Notation, utilisé très couramment par les APIs de sites web, et exports structurés de données depuis le web) correspond à cette manière de définir des objets. Nous allons y revenir plus tard dans ce chapitre.

**Accès aux propriétés d’un objet**

Il existe deux manières d’accéder à la valeur d’une propriété d’objet :

1. La notation pointée, ex: objet.prop
2. L’usage des crochets, ex: objet[prop]

Comme pour les tableaux, l’adressage par crochets a un avantage intéressant: il permet d’accéder à un élément dont la clé est stockée dans une variable.

Exemple:

let objet = {

prop1: 'a',

prop2: 'b',

prop3: 4,

};

// adressage littéral par notation pointée:

objet.prop2; // => 'b'

// adressage littéral par crochets:

objet['prop2']; // => 'b'

// adressage symbolique par crochets:

var cle = 'prop2';

objet[cle]; // => 'b'

La notation pointée est plus concise et lisible, mais impose de mentionner la clé “en dur” (littéralement) dans le code:

// adressage littéral, avec notation pointée:

objet.prop2; // => 'b'

// adressage symbolique, impossible avec notation pointée:

var cle = 'prop2';

objet.cle; // => undefined, car il n'y a pas de propriété appelée 'cle' dans l'objet

**Modification de propriétés**

La modification d’une propriété d’objet fonctionne similairement à la modification d’un élément de tableau : il suffit d’effectuer une affectation après avoir adressé la valeur à modifier.

let tableau = ['a', 'b', 4];

tableau[1] = 3; // => tableau = ['a', 3, 4]

let table = {

prop1: 'a',

prop2: 'b',

prop3: 4,

};

table.prop2 = 3; // => table: { prop1: 'a', prop2: 3, prop3: 4 }

Il est bien sûr possible d’utiliser indifféremment l’adressage par notation pointée ou par crochets, pour une affectation.

**Clés riches**

Contrairement aux restrictions imposées pour nommer les variables en JavaScript, il est possible d’inclure des espaces et des caractères spéciaux dans les noms de propriétés.

Par contre, il faut dans ce cas utiliser des apostrophes (ou guillemets) pour définir et adresser ces propriétés:

Let mesAmis = {

'Luke Skywalker': true,

'Dark Vador': false,

};

mesAmis.LukeSkywalker; // => undefined

mesAmis.Luke Skywalker; // => /!\ syntax error

mesAmis['Luke Skywalker']; // => true

**Notes sur les types**

Comme pour les tableaux, la valeur d’une propriété d’objet peut être de n’importe quel type: string, number, boolean, null, undefined, object ou function.

A noter que, comme les objets, les tableaux sont aussi de type object.

typeof { prop: 'a' }; // => "object"

typeof [ 'a', 3, 4 ]; // => "object"

Mais les tableaux ont des caractéristiques particulières que n’ont pas les objets. Notamment l’indexation numérique, des fonctions spécifiques comme slice, etc…

**Exercice 8 Annuaire téléphonique**

**Objets, usage avancé**

Comme il est possible de stocker un objet comme valeur d’une propriété d’objet, cela veut dire que nous pouvons définir des hiérarchies / arbres d’objets.

Exemple:

var compteFacebook = {

amis: {

'Luke Skywalker': true,

'Dark Vador': false,

},

groupes: {

maitresJedi: {

titre: 'Groupe des maîtres Jedi',

membres: [ 'Yoda', 'Obi Wan' ],

},

lolcats: {

titre: 'Vive les chats !',

membres: [ 'Patrick' ],

},

},

};

Dans l’exemple ci-dessus, nous avons jusqu’à trois niveaux d’imbrication d’objets (voire quatre, si on compte les tableaux comme des objets): l’objet compteFacebook contient une propriété groupes (de type objet) contenant une propriété maitresJedi (aussi de type objet) contenant deux propriétés (de type chaîne de caractères et tableau).

Cet objet peut être représenté visuellement sous forme d’un arbre.

Pour accéder au premier élément du tableau membres de la sous-propriété maitresJedi, on doit écrire le cheminement à suivre, de propriété à sous-propriétés.

Exemple:

// adressage par notation pointée:

compteFacebook.groupes.maitresJedi.membres[0]; // => 'Yoda'

// adressage par crochets:

compteFacebook['groupes']['maitresJedi']['membres'][0]; // (idem)

Dans cet exemple aussi, on peut utiliser indifféremment la notation pointée ou les crochets, pour accéder à la propriété d’un objet.

**Énumérer les propriétés d’un objet**

Il existe deux manières d’énumérer les propriétés d’un objet:

1. utiliser une boucle for-in;
2. ou itérer sur le tableau des clés de l’objet, à l’aide de la fonction Object.keys().

Une boucle for-in s’utilise de cette façon:

var mesAmis = {

'Luke Skywalker': true,

'Dark Vador': false,

};

for (ami in mesAmis) {

console.log(ami, '->', mesAmis[ami]);

}

// => lignes affichées dans la console:

// Luke Skywalker -> true

// Dark Vador -> false

Contrairement à une boucle for classique (*dans laquelle on définit une instruction d’initialisation, une expression conditionnelle et une instruction d’incrémentation, séparés par des points-virgules, pour rappel*), la boucle for-in va répéter la liste d’instructions entre accolades pour chaque clé de l’objet.

Dans notre exemple de code ci-dessus, la variable ami prendra donc la valeur d’une clé de l’objet mesAmis, pour chacune de ses propriétés.

A noter que cela fonctionne aussi sur les tableaux, sauf que l’indice de chaque élément du tableau sera fourni (au lieu de la clé de chaque propriété de l’objet).

**Énumérer les propriétés d’un objet, une autre manière**

Lorsqu’elle est appelée en passant un objet en paramètre, la fonction Object.keys() retourne un tableau contenant les clés/noms de chaque propriété de cet objet.

var mesAmis = {

'Luke Skywalker': true,

'Dark Vador': false,

};

var cles = Object.keys(mesAmis); // => [ 'Luke Skywalker', 'Dark Vador' ]

On peut alors utiliser une boucle for plus classique pour itérer sur les valeurs de ce tableau:

for (var friend= 0; friend < amis.length; friend++) {

var cle = cles[i];

console.log(friend, ':', ami, '->', mesAmis[cle]);

}

// => lignes affichées dans la console:

// 0, Luke Skywalker -> true

// 1, Dark Vador -> false

**Suppression d’une propriété**

Pour supprimer une propriété d’un objet, il faut utiliser le mot clé delete de la manière suivante:

let objet = {

prop1: 'a',

prop2: 'b',

prop3: 4,

};

Delete objet.prop2;

// => objet === { prop1: 'a', prop3: 4 }

**Exercice Répertoire téléphonique**

**Les classes dans la POO**

**Programmation Orientée Objet : classes, instances et this**

Une classe est un modèle d’objet. Elle peut être instanciée, c’est à dire qu’on crée un objet (appelé *instance*) selon ce modèle.

La modèle d’une classe consiste à assurer que chaque objet instance de cette classe aura les mêmes:

* propriétés; (*cf chapitre sur les types avancés*)
* et méthodes: des fonctions qui s’appliquent à une instance donnée.

À noter que:

* chaque instance d’une classe aura les mêmes propriétés, mais la valeur de celles-ci pourra être différente pour chaque instance;
* chaque instance d’une classe aura les mêmes méthodes, mais l’exécution de la fonction correspondante ne s’appliquera qu’à l’instance sur laquelle elle aura été appelée.

Comme nous allons le voir dans la suite du cours, les classes sont très utilisées pour manipuler la structure de pages Web. Notamment pour intégrer plusieurs instances d’un même composant sur une même page.

**Comment instancier une classe en JavaScript/ES6**

En guise d’exemple, supposons qu’on veuille intégrer un composant permettant d’afficher une galerie d’images sur notre page Web.

Supposons que ce composant soit défini par une classe nommée Galerie.

Comme pour toute classe, on peut instancier une Galerie en appelant son constructeur avec le mot clé new. Le constructeur de cette classe prend un paramètre: conteneur, une portion de la page Web dans lequel la galerie s’intègrera.

Enfin, supposons que la classe Galerie fournisse des méthodes qui seront rattachées à chaque instance de cette classe:

* ajouterImage() permet de spécifier l’URL d’une image qui sera à afficher dans cette galerie,
* et regenerer() permet de mettre à jour l’affichage de la galerie, après y avoir ajouté des images.

Voici un exemple d’instanciation de cette classe:

// supposons que conteneur référence un <div> de la page

let maGalerie = new Galerie(conteneur);

maGalerie.ajouterImage('img7.jpg');

maGalerie.regenerer();

Le mot clé **new** permet d’instancier notre classe, et donc d’exécuter son constructeur en fournissant une valeur pour le paramètre conteneur. Comme pour une fonction, l’appel au constructeur retourne l’instance de Galerie fraichement créée.

**Comment définir une classe en JavaScript/ES6**

Afin de permettre l’instanciation de la classe Galerie, le créateur du composant a dû la définir de la manière suivante :

class Galerie {

// définition du constructeur de la classe Galerie

constructor(conteneur) {

this.conteneur = conteneur;

this.urlImages = [];

}

// cette méthode permet d'ajouter une image à cette galerie

ajouterImage(url) {

this.urlImages.push(url);

}

// cette méthode permet de générer et d'afficher cette galerie dans la page

regenerer() {

var html = '';

// génération des éléments <img> dans le conteneur

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

html = html + '<img src="' + this.urlImages[i] + '" class="hidden">;';

}

this.conteneur.innerHTML == html;

}

}

À noter:

* le paramètre conteneur du constructeur de la classe a été affecté comme propriétés d’un certain objet this. (nous allons expliquer ça plus bas)

**Usage de this**

Quand on mentionne this dans la définition d’une méthode, ce mot clé représente l’instance depuis laquelle la méthode a été appelée.

Par exemple:

class Article {

constructor(titre) {

this.titre = titre;

}

getTitre() {

return this.titre; // this === article1 ou article2, dans notre exemple

}

}

var article1 = new Article('Trump élu président');

var article2 = new Article('Macron se présente');

article1.getTitre(); // => retourne 'Trump élu président'

article2.getTitre(); // => retourne 'Macron se présente'

À noter qu’en JavaScript, this est en fait utilisable depuis toute fonction, qu’elle soit ou pas définie dans une classe. Il faut retenir que l’usage de classes permet à l’interpréteur JavaScript d’affecter automatiquement à this l’instance sur laquelle s’exécute chaque méthode.

**Exercice instanciation de classe**

**Manipuler le web avec DOM**

**1. JavaScript et le DOM**

Dans les chapitres précédents, nous avons exécuté des programmes JavaScript simples de manière interactive, depuis la console de notre navigateur.

Dans ce chapitre, nous allons voir:

* comment associer un programme JavaScript à une page web,
* et comment modifier cette page dynamiquement depuis notre programme.

**Terminologie: quelques rappels sur le Web**

* WWW: *World Wide Web*, c’est le nom donné à l’ensemble des pages liées entre elles sur Internet via le protocole HTTP (nous en reparlerons dans le chapitre suivant). La plupart de ces pages sont décrites en langage HTML.
* HTML: *HyperText Markup Language*, c’est un langage qui permet de décrire la structure et le contenu d’une page web, en utilisant des éléments (balises).
* CSS: *Cascading Style Sheets*, c’est un langage qui permet de mettre en page et styliser les éléments HTML d’une page web en leur appliquant des propriétés.
* JavaScript: c’est le seul langage qui permet de donner des instructions exécutables depuis une page web au format HTML.
* Navigateur Web / *Web Browser*: c’est un logiciel d’affichage de pages web (généralement composées de fichiers HTML, CSS et JavaScript), et permettant à l’utilisateur d’interagir avec celles-ci.
* DOM: *Document Object Model*, c’est à la fois le nom qu’on donne à l’ensemble des éléments qui constituent une page Web ouverte dans le navigateur (telle qu’elle est structurée en mémoire), et à l’API qui permet de manipuler ces éléments.
* API: *Application Programming Interface*, est un ensemble de fonctions fournies par un logiciel (par exemple: un navigateur Web, ou un serveur Web) qui permettent à d’autres programmes d’interagir / échanger des informations avec lui.

**Associer un programme JavaScript à une page web**

Vous devez savoir qu’une page web peut être associée à une feuille de style CSS. Pour cela, le code source HTML de cette page doit contenir un élément <link> donnant l’URL du fichier CSS correspondant.

De la même façon, pour associer un programme JavaScript à une page web, il suffit d’ajouter un élément <script> donnant l’URL du programme en question (dont le fichier porte généralement l’extension .js).

Exemple de code source HTML d’une page web:

<!DOCTYPE html>

<html>

<head>

<link rel="stylesheet" href="style.css" />

</head>

<body>

<h1>Bonjour !</h1>

</body>

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

</html>

Trois choses importantes à remarquer:

* l’élément <script> peut être défini dans le <head> ou aprés le <body>, mais il est généralement recommandé de l’insérer juste aprés la fin du <body>;
* l’URL du script doit être fournie via l’attribut src;
* mais surtout, contrairement aux éléments <link>, les éléments <script> ne doivent pas être exprimés sous forme d’une balise auto-fermante (finissant par />) => il est impératif d’**utiliser une balise fermante </script> après chaque balise ouvrante <script>**.

Les scripts ainsi intégrés dans le <body> de la page seront exécutés au fur et à mesure qu’ils sont découverts et chargés par le navigateur.

Remarque, il est aussi possible d’intégrer directement notre programme JavaScript entre les balises <script> et </script>, pour éviter de le stocker dans un fichier séparé. Cette méthode n’est pas recommandée car elle peut causer des erreurs de syntaxe. Cette méthode est à proscrire désormais on écrit du Javascript dans un fichier à part et on le rappel en indiquant son chemin.

**Exercice 9 Dire bonjour au monde**

### Accéder aux éléments de la page Web depuis JavaScript

Les navigateurs Web (tels que Google Chrome) donnent accès à une API (voir définition plus haut) permettant à nos programmes JavaScript d’interagir avec le DOM de la page Web à laquelle ils sont liés.

C’est à dire qu’un script intégré dans une page peut utiliser des fonctions permettant de manipuler le contenu de cette page. Par exemple: pour récupérer des informations saisies par l’utilisateur dans des champs de la page, ou encore modifier le contenu et/ou l’affichage de la page.

Pour accéder à un élément de la page, il faut identifier précisément (c.a.d. sans ambiguïté) cet élément auprès du navigateur. Par exemple: en l’adressant par son identifiant unique (attribut id de l’élément).

Pour cela, l’API du DOM met à notre disposition un objet (cf chapitre précédent) appelé document, et cet objet contient plusieurs fonctions. Nous allons d’abord nous intéresser à la fonction getElementById() qui permet d’accéder à l’objet représentant un élément de la page, en fonction de son identifiant unique.

Supposons que notre page Web contienne les éléments suivants:

<body>

<p id="premier-paragraphe">Bonjour</p>

<p id="deuxieme-paragraphe">le monde</p>

</body>

Nous pouvons alors accéder au premier paragraphe en JavaScript (ex: depuis un script rattaché à cette page, ou depuis la console du navigateur) de la manière suivante:

document.getElementById('premier-paragraphe');

// => retourne un objet qui représente l'élément HTML correspondant

En exécutant cet appel de fonction dans la console, on voit s’afficher ce qu’elle retourne : un objet JavaScript qui représente l’élément HTML ayant premier-paragraphe comme identifiant.

Nous allons voir plus bas que l’objet document fourni par l’API du DOM contient d’autre fonctions permettant d’accéder à des éléments, comme getElementsByClassName() ou encore querySelector().

#### Application:

1. allez sur le site web de votre choix,
2. utilisez l’onglet “Éléments” de Chrome Dev Tools (barre latérale dans laquelle se trouve aussi la console JavaScript),
3. repérez un élément qui possède un attribut id,
4. dans la console, utilisez getElementById() de manière à afficher l’objet JavaScript représentant cet élément.

Astuce: pour accéder plus rapidement à la partie du DOM qui représente un élément d’une page Web, effectuer un clic-droit sur cet élément de la page, puis cliquez sur “Inspecter”. Vous vous retrouverez immédiatement dans l’onglet “Éléments” de la page, à l’endroit où est défini cet élément.

### Récupérer la valeur d’un champ de saisie <input>

Maintenant que nous savons accéder à l’objet JavaScript correspondant à un élément HTML de la page, nous allons voir comment récupérer des données de cet élément.

Pour cela, nous allons utiliser la propriété value de l’objet JavaScript représentant un élément HTML.

Imaginons une page HTML contenant le formulaire suivant:

<body>

<form>

<label for="nom">Nom:</label>

<input id="nom" value="Michel" />

<label for="prenom">Prénom:</label>

<input id="prenom" value="Jean" />

</form>

</body>

Pour accéder à l’objet JavaScript représentant le champ portant l’identifiant nom, nous allons utiliser la fonction getElementById() de cette manière:

document.getElementById('nom');

Enfin, pour récupérer la valeur actuelle de ce champ, il suffit d’utiliser la propriété value de cet objet:

document.getElementById('nom').value;

Dans notre exemple, l’exécution de cette instruction JavaScript retournera la valeur du champ correspondant: "Michel".

Si on exécute à nouveau cette instruction après que l’utilisateur a modifié la valeur du champ, la valeur retournée correspondra à la valeur actuelle du champ (après modification).

**Exercice 9 Récupération des valeurs du formulaire**

**Réagir aux actions de l’utilisateur sur la page**

Maintenant que nous savons accéder aux données d’une page HTML depuis un programme JavaScript, et faire en sorte que ce programme s’exécute au chargement de la page, nous allons voir comment exécuter des instructions JavaScript en réponse à une action de l’utilisateur sur la page.

À chaque fois que l’utilisateur interagit avec une page Web, le navigateur déclenche des *événements*. Ces événements sont mis à disposition par l’API du DOM, afin qu’un programme JavaScript puisse les intercepter et réagir à certains d’entre eux.

Quelques exemples d’événements :

* click: l’utilisateur a cliqué sur un élément
* change: l’utilisateur a changé la valeur d’un champ de saisie
* mouseover: l’utilisateur a survolé un élément à l’aide de la souris

On peut définir le comportement (la réaction) que doit adopter le navigateur lorsqu’un événement survient, en y associant une fonction JavaScript.

Par exemple, nous pourrions définir une fonction direBonjour() contenant alert('bonjour !'), puis demander au navigateur d’appeler cette fonction à chaque fois que l’utilisateur clique sur un bouton.

Il existe plusieurs moyens d’intercepter des événements en y attachant une fonction :

* la fonction addEventListener() (que nous verrons peut-être plus tard)
* et les propriétés on\* associées à chaque nom d’événement, ex: onclick, onchange, onmouseover…

Pour l’instant, nous allons employer la méthode la plus simple: affecter une fonction à la propriété d’un élément correspondante à l’événement choisi.

Imaginons une page HTML contenant un bouton:

<body>

<button id="mon-bouton">Mon Beau Bouton</button>

</body>

Pour afficher un alert à chaque fois que l’utilisateur cliquera sur ce bouton (événement click), nous devons affecter une fonction à la propriété onclick de l’objet JavaScript représentant ce bouton:

document.getElementById('mon-bouton').onclick = function direBonjour() {

alert('bonjour !');

};

**Exercice 9 Calculatrice**

**Manipulation des classes**

En HTML, chaque élément peut être associé à des classes. On les spécifie en les énumérant dans l’attribut class de l’élément.

Exemple:

<p id="premier">texte visible</p>

<p id="second" class="hidden">ce texte devrait être caché</p>

Dans cet exemple, l’élément <p> est associé à une classe hidden. Pour que tous les éléments associés à cette classe soient cachés, il suffirait alors de définir la règle CSS suivante:

.hidden {

display: none;

}

En JavaScript, il est possible de modifier dynamiquement la liste de classes associées à un élément (et donc son affichage, en fonction des règles CSS qui sont associées à ces classes), grâce à la propriété [classList](https://developer.mozilla.org/fr/docs/Web/API/Element/classList), et à ses fonctions add() et remove().

Exemple:

var element = document.getElementById('un-element');

element.classList.add('hidden'); // => la classe hidden va être associée à l'élément

**Exercice : 10 Afficher/cacher**

## 2. Accéder à des éléments par class

Dans la partie précédente, nous avons utilisé la fonction getElementById() pour accéder à un des éléments de notre page, à partir de son identifiant id.

L’API du DOM donne également accès à une fonction qui permet d’accéder à l’ensemble des éléments qui portent une même classe class: getElementsByClassName().

Supposons que nous soyons sur une page HTML contenant plusieurs paragraphes portant une même classe:

<p>C'est l'histoire d'un garçon troublé</p>

<p class="spoiler">car il s'avère que son ours en peluche a disparu</p>

<p class="spoiler">mais finalement il le retrouve</p>

Pour accéder à tous les éléments portant la classe spoiler, nous allons utiliser le code JavaScript suivant:

var elements = document.getElementsByClassName('spoiler'); // => tableau d'éléments

À noter que, contrairement à getElementById(), cette fonction ne retourne pas un seul élément, mais un tableau de plusieurs éléments.

Pour effectuer une même manipulation sur chacun des éléments de ce tableau, il suffit d’utiliser une boucle for, tel que nous l’avons vu dans un précédent chapitre:

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

var element = elements[i]; // i-ème élément du tableau d'éléments.

### Exercice : 10 Cacher les spoilers

**Manipulation de style CSS**

Nous savons à présent :

* accéder à des éléments HTML, à partir de leur id ou class;
* et modifier la liste des classes associées à un élément HTML.

Dans cette partie, nous allons voir qu’il est possible de modifier dynamiquement le style d’éléments HTML sans avoir à manipuler de classes CSS.

De la même manière que les éléments HTML <input> fournissent une propriété JavaScript value permettant d’accéder à leur valeur, tous les éléments HTML fournissent une propriété JavaScript style.

Alors que la propriété value est de type String (chaîne de caractères), la propriété style est de type Object.

L’objet associé à la propriété style est structuré similairement à une règle CSS: il est constitué de propriétés clé-valeur.

Exemple de règle CSS :

#mon-element {

border: 1 solid black;

background-color: red;

}

… et l’objet JavaScript correspondant, tel qu’associé à la propriété style de l’élément :

{

border: '1 solid black',

backgroundColor: 'red'

}

Il y a deux **différences importantes** à noter :

* la propriété style de l’élément doit respecter la syntaxe d’objet JavaScript (JSON), donc les valeurs de propriétés sont des chaînes de caractères (entre apostrophes), et les propriétés doivent être séparées par des virgules (au lieu des point-virgules de la notation CSS);
* enfin, les noms de propriétés contenant des tirets doivent être écris en respectant la notation “camel case” (pour rappel: mots collés, avec majuscules en début de chaque mot sauf le premier).

**Exemple de changement de style**

Prenons le fichier HTML suivant :

<p id="premier">texte important</p>

<p id="second">autre texte</p>

Pour modifier la couleur de fond du premier paragraphe, il faut exécuter le code JavaScript suivant:

var element = document.getElementById('premier');

element.style.backgroundColor = 'red' ;

**Exercice 10 Surbrillance au clic**

**4. Accéder à des éléments par nom d’élément**

**Fonction DOM: getElementsByTagName()**

Jusqu’ici, nous avons vu deux fonctions de sélection d’éléments du DOM:

* getElementById() retourne un élément, quand on fournit son id;
* et getElementsByClassName() retourne un tableau d’éléments, quand on fournit leur class.

Dans certains cas, il est pratique d’adresser les éléments par type (nom de balise). C’est ce que permet la fonction getElementsByTagName().

Exemple:

var images = document.getElementsByTagName('img');

// => images est un tableau contenant tous les éléments <img> de la page

À noter que, comme getElementsByClassName(), getElementsByTagName() retourne un tableau d’éléments.

**5. Manipulation d’attributs**

**Fonctions DOM: getAttribute() et setAttribute()**

Nous avons vu qu’il était possible de récupérer et/ou modifier la valeur d’un champ <input> à l’aide de la propriété value de l’objet JavaScript représentant ce champ.

Il se trouve que value est aussi le nom de l’attribut HTML correspondant. En effet, le DOM donne accès à des propriétés correspondant aux attributs standards des éléments HTML.

En guise d’exemple, imaginons l’élément HTML suivant:

<button id="mon-bouton" data-numero="5">

L’attribut data-numero n’est pas standard, mais nous avons le droit de nous en servir pour rattacher des informations à cet élément.

Pour récupérer la valeur de cet attribut, utilisons la fonction getAttribute() de la manière suivante:

document.getElementById('mon-bouton').getAttribute('data-numero');

// => cet appel de fonction retournera 5

Pour modifier la valeur de cet attribut, utilisons la fonction setAttribute() de la manière suivante:

document.getElementById('mon-bouton').setAttribute('data-numero', 7);

// => l'attribut va prendre la valeur 7, au lieu de 5

Un petit plus il existe la methode querySelector() qui permet de renvoyer le premier élément enfant qui correspond à un élément spécifié selecteur css d’un élément(classe)

Exemple :

<div id="querySelector">

<h2 class="changeText">Alors Javascript </h2>

<p class="changeText">java ou pas ?</p>

</div>

<p>Cliquez sur le bouton et vous aurez votre réponse .</p>

<button onclick="myFunction()">Reponse ici</button>

<script>

function myFunction() {

var x = document.getElementById("querySelector ");

x.querySelector(".changeText ").innerHTML = "Non c’est easy!";

}

</script>

Il existe également la méthode querySelectorAll() qui renvoie tous les éléments qui correspondent à un ou à plusieurs sélecteurs Css.

Ce sont des méthodes très couramment utilisées donc ne les laissez pas de côté.

**6. Manipulation de contenu HTML d’un élément**

**Propriété innerHTML**

Avant que nous décrivions comment modifier la structure du DOM d’une page web, sachez qu’il est possible de modifier directement depuis JavaScript le code HTML contenu par un élément HTML: à l’aide de la propriété innerHTML de cet élément.

Exemple:

<!-- Avant: -->

<p id="mon-parag">texte initial</p>

document.getElementById('mon-parag').innerHTML = 'image: <img src="image.jpg">';

Cette affectation modifiera le DOM de la page web de la manière suivante:

<!-- Après: -->

<p id="mon-parag">image: <img src="image.jpg"></p>

**7. Scope et génération de fonctions**

Nous avons vu plus haut qu’il fallait utiliser une boucle for pour effectuer une même opération sur un ensemble d’éléments HTML.

Exemple:

// supposons que elements soit un tableau d'éléments HTML

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

elements[i].classList.add('bordure-rouge');

}

Ce genre de code ne pose aucun problème.

Par contre, dès qu’une boucle for définit une fonction qui sera appelée plus tard et dépend de la variable de boucle, notre programme se comporte de manière inattendue…

C’est notamment le cas si on définit une fonction onclick dans une boucle, tel que dans l’exemple suivant:

// supposons que elements soit un tableau d'éléments HTML

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

elements[i].onclick = function() {

alert('vous avez cliqué sur l\'élément nº' + i);

}

}

En cliquant sur les éléments sur lesquels s’applique cette boucle, vous constaterez que l’alert affichera toujours le même élément, alors que la variable de boucle i prend bien comme valeur l’indice de chaque élément de la page.

Ce comportement est dû à la manière dont JavaScript référence les variables: leur “portée” (appelée “scope”, en Anglais).

**“Scope”: portée des variables**

**En JavaScript, toute variable définie par var/let et rattachée à la fonction qui contient sa définition, ou dans l’espace “global” sinon.**

Illustration:

var variableGlobale = 4;

function maFonction() {

let variableLocale = 5;

console.log('(maFonction) variableGlobale:', variableGlobale); // => 4

console.log('(maFonction) variableLocale:', variableLocale); // => 5

}

maFonction();

console.log('variableGlobale:', variableGlobale); // => 4

console.log('variableLocale:', variableLocale); // => ReferenceError: variableLocale is not defined

Quand on mentionne une variable à l’intérieur d’une définition de fonction, JavaScript va d’abord chercher s’il existe une variable localement définie, puis chercher dans les contextes parents dans lesquels ont été définis cette fonction, jusqu’à l’espace global, si besoin.

Dans le cas de notre boucle for (cf exemple plus haut), la fonction affectée à la propriété onclick utilise la variable i, et cette variable n’est pas locale à cette fonction: elle est rattachée au contexte parent (celui qui contient la définition de fonction).

Du coup, lorsque cette fonction est appelée (en l’occurrence: au moment où l’utilisateur clique sur un élément), **c’est la valeur de cette variable *au moment de l’appel* qui va être utilisée**. Or, au moment de l’appel, notre boucle a fini d’itérer, et la variable i vaut donc sa valeur maximum. (en l’occurrence: elements.length)

Pour éviter ce problème de portée, le plus simple est d’appeler à chaque itération une fonction en passant notre variable i en paramètre, de manière à ce que sa valeur soit rattachée à la fonction (comme si c’était une variable locale), et non au contexte parent.

Il existe deux moyens classiques d’appliquer cette pratique:

1. la **closure**: définir une fonction anonyme et l’appeler dans la foulée;
2. ou appeler une fonction qu’on aura définie à l’extérieur de notre boucle.

Dans les deux cas, cette fonction devra retourner une autre fonction, de manière à ce que cette dernière ne soit pas appelée immédiatement au moment de la définition de la fonction parente.

**Fonction génératrice de fonction**

Nous avons vu qu’une fonction pouvait retourner une valeur de n’importe quel type, et que les fonctions étaient un des types avancés du langage JavaScript. Il est donc tout à fait possible qu’une fonction retourne une autre fonction !

Exemple :

function mere() {

console.log ('fonction mere appelée');

return function fille() {

console.log ('fonction fille appelée');

};

}

let fct = mere(); // => affiche 'fonction mere appelée' puis retourne la fonction fille

// => la fonction fille est affectée à la variable fct

fct(); // => affiche 'fonction fille appelée'

Pour éviter le problème de portée expliqué plus haut, il suffit alors de passer la valeur en paramètre de la fonction mère, qui sera appelée à chaque itération de la boucle:

function mere(i) {

// i est \*promue\* comme variable locale à la fonction mere

// => à chaque appel de la fonction qu'elle retourne, elle aura conservé sa valeur

return function fille() {

alert('vous avez cliqué sur l\'élément nº' + i);

// i fait référence à la valeur qui avait été passée en paramètre

// de l'appel à la fonction mere(), à chaque itération de boucle.

};

}

// supposons que elements soit un tableau d'éléments HTML

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

elements[i].onclick = mere(i);

// => la fonction retournée par mere va être affecté à fille