IFT215

Interfaces et multimédia

Laboratoire 4

Équipe : 2 personnes

Enseignant :	Frédéric Bergeron
Réception de l'énoncé :	2 novembre 2021
Remise du devoir :	7 novembre 2021

Objectif

Introduction à la programmation d'interface dans un environnement web.

Description

Ce travail constitue la troisième partie de quatre se rapportant à la programmation d'interfaces web. Ce troisième travail se veut la suite de l'introduction aux langages HTML et CSS.

Mise en situation

Vous travaillez pour une entreprise qui fait du développement web en utilisant les technologies HTML, CSS et JavaScript. Votre client actuel est une compagnie de suppléments alimentaires pour athlètes. Votre tâche aujourd'hui est d'implémenter certains éléments d'interaction du site web. Vous devez vous rapprocher le plus possible des différentes captures d'écran parsemant ce document.

Évaluation

Ce travail de laboratoire compte pour 1% de votre session. La remise a lieu sur Turnin. Le point est réparti comme suit :

1 point : Fonctionnalités identiques à celles présentées en fin des sections 3, 4 et 5.

Travail dirigé

Cette partie est un mélange entre un tutoriel et un travail pratique. Ce travail tient pour acquis que les parties 1 et 2 ont été complétées. Nous allons étendre le site web construit.

1. Javascript

Javascript est le langage principal, pour ne pas dire unique, d'interactivité dans les clients web. Le langage a été créé en 1995 et est maintenant disponible sur virtuellement tous les navigateurs. C'est un langage complet et complexe dont nous ne verrons que les aspects de base. Il n'y a pas de lien entre Javascript et Java, hormis que les noms ont été harmonisé pour profiter de la popularité de l'un ou de l'autre. La syntaxe peut parfois être semblable, mais le langage reste fondamentalement différent. Officiellement, Javascript est une implémentation parmi d'autres de la norme ECMAScript, mais sa domination face aux autres fait qu'on oublie généralement cette distinction. Tout comme HTML et CSS, Javascript a essentiellement abandonné le concept de version pour plutôt intégrer les mise-à-jour dès que possible, une à la fois. La version de référence en cours est ECMAScript 2020 et une nouvelle version de la norme est prévue en 2022.

Tout comme Java, il existe plusieurs interpréteurs pour Javascript. Historiquement, chaque navigateur avait son propre interpréteur, mais la tendance est à la fusion. Les principaux sont maintenant ceux de Google et de Firefox, les autres navigateurs utilisant les versions libres de ceux-ci. Contrairement à Java toutefois, les mise-à-jours ne sont font pas à date fixe, donc les fonctionnalités n'arrivent pas en même temps dans tous les navigateurs. Il faut donc vérifier la compatibilité avant d'utiliser les éléments spécialisés. Nous n'aurons pas de problème ici. La même limitation est aussi vraie pour HTML et CSS. Le site https://caniuse.com/ offre une ressource claire pour rapidement déterminer si un élément est disponible dans le navigateur visé.

L'histoire n'étant pas le sujet de ce cours, passons maintenant aux aspects techniques. Javascript supporte plusieurs paradigmes : impératif, objet et fonctionnel. Il est malheureusement commun de rencontrer les trois paradigmes en même temps. À la manière de Python, Javascript est faiblement typé, ce qui signifie que les variables n'ont pas de type officiel, mais tout de même un type interne qui sera converti au besoin vers un autre type interne. Cela apporte beaucoup de confusion et de comportements étranges (en voici quelques-uns : https://dev.to/damxipo/javascript-versus-memes-explaining-various-funny-memes-2o8c).

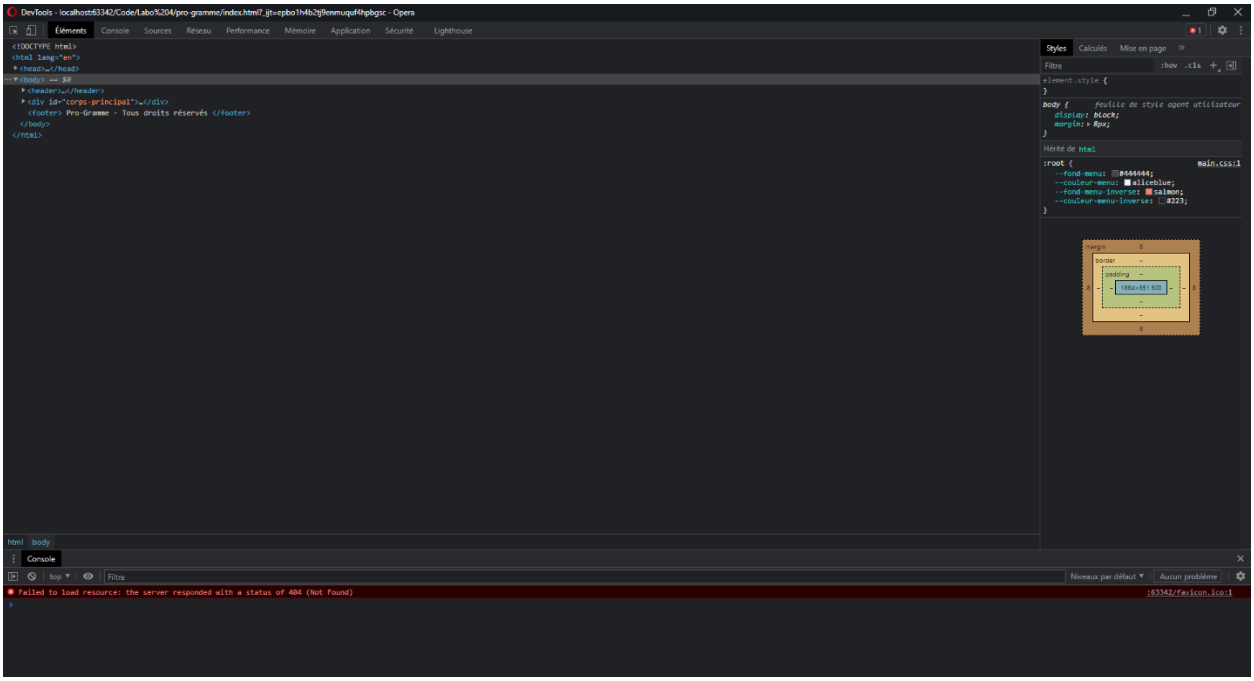
Variables

En Javascript, les variables n'ont pas de type à la déclaration. Il y a 3 mots-clés pour déclarer des éléments : var, let et const. var n'a plus vraiment de raison d'exister et est maintenant remplacé par let. Prenez toujours let, sauf si vous savez ce que vous faites. let est donc le mot-clé pour déclarer une variable avec la portée habituelle (le bloc défini par des accolades {}). const est le mot-clé pour déclarer une constante.

```
let a = 5;
let b = "Tomate";
let c = true;
let d = null;
```

Voici donc à quoi ressemble la déclaration. Aucune de ces variables n'a de type, le même mot-clé s'applique à toutes. Toutefois, elles ont toutes un type interne différent. Dans l'ordre, nous avons un number, un string, un boolean et un null.

Pour faire des tests rapidement, vous pouvez ouvrir la console de votre navigateur. Allez d'abord sur une page de notre pro-gramme, puis faites un clic droit et sélectionner « Inspecter ». Voici la mienne sur Opera :



Dans le bas, il y a la console. Il s'agit d'un interpréteur Javascript, vous pouvez y écrire n'importe quel code et il va s'exécuter dans votre page web courante. Copiez-y les quatre déclarations ci-haut puis faites « Entrer ». Vos variables existent maintenant dans l'environnement. Pour en afficher une, simplement écrire son nom puis faire « Entrer ». Nous pouvons aussi faire des opérations. Par exemple, nous pouvons faire a + b. Quel sera le résultat d'un chiffre plus une chaîne de caractères? Seul javascript le sait! En vrai, la réponse est « 5Tomate ». Javascript a donc automatiquement converti le chiffre 5 vers le caractère '5' puis a concaténé les deux chaînes. Essayez différentes combinaisons, c'est rigolo. Vous pouvez aussi assigner le résultat à une nouvelle variable. Par exemple, nous pourrions écrire « f = a + b ». Nous avons maintenant la variable f qui existe. Eh oui, f existe même si elle n'a pas été déclarée avec let. Il est recommandé de toujours déclarer les variables, mais ce n'est pas obligatoire. Dans le même ordre d'idée, il n'est pas obligatoire de mettre un ; à la fin des énoncés, mais c'est recommandé.

Fonction

Javascript permet la création de fonctions, à la manière de C++. Le mot-clé function est celui utilisé. Javascript n'étant pas typé, il n'y a donc pas de type de retour. Je peux donc déclarer une fonction ainsi :

```
function maFonction(param1, param2){

}
```

maFonction est le nom de la fonction. param1 et param2 sont alors les paramètres de ma fonction. Je peux l'appeler de cette façon :

```
maFonction(a,b);
```

Les paramètres sont passés par valeur. (Pour une discussion plus complète, voir https://stackoverflow.com/questions/518000/is-javascript-a-pass-by-reference-or-pass-by-value-language).

Il est aussi possible d'assigner une fonction à une variable :

```
let g = function (param1, param2){}

g(a,b);
```

Cela complexifie les choses! Cela veut donc dire qu'il est possible de passer des fonctions en paramètres à une autre fonction. Et là les choses peuvent commencer à être rapidement complexe. Donc passons.

Objet

Nous avons vu que Javascript supporte le paradigme orienté-objet. C'est vrai, mais fait différemment qu'en Java, par exemple. Javascript était à l'origine basé sur le concept de prototype, un modèle objet dans lequel les objets d'une classe peuvent acquérir de nouvelles fonctionnalités indépendamment des autres. Pour cette raison, il importe de vraiment distinguer les objets et les classes comme étant deux choses séparées.

De même, il est possible de déclarer des objets qui ne sont liés à aucune classe. (!!!!) C'est de là que viens JSON, le JavaScript Object Notation. Un tel objet se déclare comme une variable, mais dont la valeur est alors un objet JSON.

```
let js = { 'a': 1, 'b': 'tomate'}
```

js.a affiche alors 1 dans la console. De ce fait, un objet Javascript est donc semblable à un dictionnaire public, un ensemble clé-valeur. Ce genre d'objet n'inclus aucune méthode directement (bien qu'une clé de l'objet puisse avoir pour valeur une fonction).

Classes

Javascript offre aussi un modèle de classe plus conforme à ce que vous avez vu dans d'autres cours. La syntaxe est plutôt simple :

```
class maClasse {
    constructor(param) {
        this.p = param;
    }

    uneMethode(param){
        this.p += param;
    }
}
```

```
let obj = new maClasse(5);
obj.uneMethode(6);
```

Le mot-clé class permet de définir une classe. Le constructeur de la classe est nommé constructor. On défini une méthode en donnant un nom, suivit des paramètres, suivit du corps de la méthode. On se crée un objet de la classe avec le mot-clé new. Le mot-clé this permet de référer à l'objet, pour obtenir ses attributs ou ses méthodes.

2. Inclusion dans un page web

Tout comme CSS, il y a plusieurs façons d'inclure du Javascript dans une page web. En fait, les choix sont essentiellement les mêmes. Le premier choix est l'inclusion directe, cette fois-ci avec la balise <script>.

```
<script>
    let a = 6;
</script>
```

Cette balise peut être placée n'importe où dans votre page. Il est toutefois plus typique de les retrouver dans l'en-tête (le <head>) ou alors en fin de page. Ce choix est habituellement basé sur des besoins d'optimisations (voir IFT604 pour une discussion approfondie sur le sujet).

Naturellement, la seconde option est de placer notre code Javascript dans un fichier externe.

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

Ce sera souvent la méthode préférée pour faciliter la séparation du code. Au même titre que nous séparons le contenu de l'affichage, nous voudrons aussi séparer l'interaction du contenu. C'est un peu l'idée du MVC qui revient dans les composants centraux du web.

Il est possible d'intégrer plusieurs fichiers externes, pouvant provenir de sources diverses.

```
<script src="main.js"></script>
<script src="js/page.js"></script>
<script src="ext.js"></script>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js">
</script>
```

Ici, j'ai donc un premier script nommé main.js situé dans le même répertoire que la page html. J'ai aussi un fichier page.js situé dans un dossier nommé js situé dans le répertoire courant. En dernier, j'ai un script externe situé sur un autre site web. Ce script n'est pas un exemple aléatoire, mais plutôt l'une des librairies parmi les plus répandues. jQuery offre une interface très simple pour manipuler les composants d'une page web. Nous utiliserons toutefois les méthodes natives dans ce tutoriel, mais vous pourrez regarder de ce côté pour explorer plus loin dans votre temps libre.

Tant qu'à être ici, nous allons créer notre premier fichier Javascript, nommé main.js. Donc, faites un clic-droit sur le projet, allez sur new, puis sélectionnez javascript file. Incluez-le maintenant dans l'en-tête (le <head>) de index.html. Dans mains.js, écrivez le contenu suivant :

```
console.log('Bonjour moi!');
```

Maintenant, exécutez la page index.html. Il n'y a aucun changement dans la page. Toutefois, si vous regardez dans la console de votre navigateur, vous verrez qu'il y est écrit « Bonjour moi! ». console.log est l'équivalent Javascript de System.out.println. Ajoutez maintenant cette ligne et exécutez :

```
alert('Bonjour toi!');
```

Cette fois-ci, une fenêtre modale apparait avec le message « Bonjour toi! ». Alert sert à afficher des fenêtres modales, la plus ennuyeuse. Confirm fait une fenêtre modale demandant une confirmation. Prompt permet quant à lui de poser une question et de recevoir la réponse (qui s'affichera dans la console) :

```
let a = prompt("Aimes-tu les carottes?")
console.log(a)
```

Remarquez que les fenêtres modales apparaissent avant le contenu de la page, qui n'arrive que lorsqu'elles sont parties. C'est parce que les scripts javascript s'exécutent dès que la balise est rencontrée, ici avant que le body ne soit lu et donc affiché. Le code est aussi exécuté parce qu'il n'est pas contenu dans une fonction. Effacez tout le contenu de main.js et ajoutez ces lignes :

```
function direBonjour(){
    alert('Bonjour toi!');
}
```

Lors de l'exécution, plus rien ne se passe. En effet, les fonctions ne s'appellent pas toutes seules! Typiquement dans le web, nous voudrons réagir à ce qui se passe dans la page web et non pas exécuter un code dès le départ. Nous voudrons toutefois souvent exécuter du code dès que la page aura fini de se charger. Dans tous les cas, cela revient à réagir à des événements, et donc faire de la programmation événementielle. C'est le sujet de la prochaine section.

3. Événements

La programmation web, tout comme la programmation d'interfaces au sens large, est largement basée sur la réaction à des événements. Javascript est conçu pour fonctionner dans ce paradigme de programmation événementielle avec simplicité. De fait, il existe deux types d'événement : ceux émis par le navigateur et ceux lancés par votre code. Les deux se traitent de la même façon (du moins, à la réception)

La liste des événements créés par le navigateur web est vaste et toujours grandissante. Une référence complète se trouve ici : https://www.w3schools.com/tags/ref_eventattributes.asp.

Naturellement, nul besoin de tous les connaître par cœur. Le plus simple est d'aller voir si ça existe lorsqu'on voudrait s'en servir. Les événements sont séparés par catégories selon leur source. On distingue ainsi les événements provenant de la souris de ceux provenant d'un formulaire.

Les événements sont associés à une balise HTML qui sert de source. On dira donc qu'on attache un événement à une balise (ou plutôt on attache un observateur d'événement à la balise). Pour ce faire, nous pouvons soit le faire directement dans le HTML, soit directement en Javascript. Nous verrons comment le faire en Javascript dans la prochaine section.

Pour attacher une réponse à un événement dans le HTML, il suffit d'ajouter un attribut à notre balise au nom de l'événement qui nous intéresse, puis à donner le code en paramètre. Par exemple, pour réagir au clic sur un bouton, nous ferions :

```
<button onclick="du code Javascript ici">Clic sur moi!</button>
```

Nous pouvons écrire le Javascript directement à cet endroit. Vous reconnaitrez toutefois qu'il ne serait pas très approprié de le faire là, en raison du principe de la séparation du code. Ainsi, nous allons plutôt appeler une fonction à cet endroit. La fonction doit alors être dans un fichier de code qui a été importé avant la balise dans le HTML (donc que le <script> est placé avant le <button>).

Nous allons maintenant commencer à améliorer notre site web. La première modification sera dans notre menu de gauche. Nous allons réagir sur le clic pour indiquer visuellement lequel a été choisi.

Tout d'abord, allons dans le fichier main.js (s'il n'est pas vide, effacez son contenu). Nous allons nous faire une fonction pour gérer le clic :

```
function clicMenuGauche(){

}
```

Maintenant, sur chacun des éléments <a> du menu gauche (ne le faites que dans index.html), nous allons ajouter un attribut onclick qui va appeler cette fonction. Si votre fichier main.js est bien déclaré, WebStorm devrait automatiquement vous suggérer la fonction dès que vous écrivez le onclick :

```
<section id="menu-gauche">
    <a href="#para-1" onclick="clicMenuGauche()">Premier paragraphe</a>
    <a href="#para-2" onclick="clicMenuGauche()">Second paragraphe</a>
    <a href="#para-3" onclick="clicMenuGauche()">Troisième paragraphe</a>
</section>
```

La fonction clicMenuGauche ne fait rien, donc impossible de savoir si cela fonctionne. Mettez-vous un message à faire apparaître dans la console.

```
function clicMenuGauche(){
    console.log('clic');
}
```

« clic » s'affiche dans la console à chaque fois que nous cliquons sur un lien du menu. Mais nous ne savons pas quel lien été cliqué, puisqu'ils utilisent tous la même fonction! Il existe plusieurs façons de récupérer la source de l'événement. Une façon simple consiste à utiliser le mot-clé this, qui permet de référencer l'objet courant :

```
<section id="menu-gauche">
    <a href="#para-1" onclick="clicMenuGauche(this)">Premier paragraphe</a>
    <a href="#para-2" onclick="clicMenuGauche(this)">Second paragraphe</a>
    <a href="#para-3" onclick="clicMenuGauche(this)">Troisième paragraphe</a>
</section>
```

Quel objet courant? Il n'y a pas d'objet, puisque nous sommes en HTML. C'est là une beauté de Javascript. Lorsque nous sommes en Javascript dans un navigateur web, le navigateur nous donne accès à un ensemble de variables globales concernant la page et son environnement. L'une de ces variables est l'objet « document » qui nous donne accès à la page. Notre page en elle-même consiste en un objet de type DOM (pour document object model). Tout comme l'objet DOM du cours IFT287, celui de Javascript nous donne accès à l'ensemble des balises et des attributs de notre document HTML. Ainsi, le this fait référence à l'objet courant auquel l'attribut onclick est attaché, donc le <a>. Avec la fonction suivante, nous pouvons donc faire afficher dans la console l'objet courant :

```
function clicMenuGauche(param){
    console.log(param);
}
```

Puisque nous avons accès à l'objet, nous pouvons aussi modifier ses propriétés directement. Nous pourrions donc changer la couleur du texte de l'élément qui a été cliqué :

```
function clicMenuGauche(lien){
    lien.style.color = "#F00";
}
```

Ici j'ai renommé la paramètre lien, pour plus de clarté. Le lien est un objet représentant une balise. J'accède donc à son attribut style puis à la clé color selon la modèle objet, en chaînant des « . ». Je pourrais aussi modifier le contenu :

```
function clicMenuGauche(lien){
    lien.style.color = "#F00";
    lien.innerText += " cliqué!"
}
```

Ici, la propriété innerText me donne accès au nœud de texte de ma balise. Je pourrais aussi avoir accès au innerHTML pour modifier les éléments à l'intérieur des limites de ma balise.

Remarquez que le comportement de ma fonction n'est pas très intelligent. Si je clique sur plus d'un lien, l'ancien ne va pas perdre sa couleur rouge. Si je clique plus d'une fois sur un lien, alors « cliqué » est ajouté à chaque fois.

C'est le comportement attendu pour cette page. Pour la page index.html, vous soumettrez ce qui a été fait dans cette section.

4. Sélection et manipulation

Le code a qui été fait à la section précédente présente de nombreux problèmes au niveau de la séparation du code, en plus d'avoir un comportement désagréable. Tout d'abord, notre Javascript modifie directement le style d'un élément, ce qui ne devrait se produire que dans le CSS. Ensuite, notre HTML inclut du code Javascript alors qu'il serait bien mieux que tout le Javascript soit séparé dans son propre fichier. Un programmeur voulant regarder uniquement ce que fait main.js n'aurait aucune idée si clicMenuGauche est jamais appelée, ni par où.

Déplacez-vous dans la page inscription pour cette partie du laboratoire. Créez un nouveau fichier Javascript nommé « inscription.js » et insérez-le dans le <head> de la page inscription.

Depuis le moment où j'ai introduit la notion du DOM, vous devez vous douter qu'en utilisant cet objet il sera possible de naviguer dans notre page par programmation pour trouver les éléments de notre choix. C'est bien le cas. L'objet document nous offre trois méthodes pour rechercher des balises dans notre page web :

1. document.getElementById(*id*) Si nous connaissons la valeur de l'attribut id de notre élément, cette méthode va nous le retourner directement. Il ne devrait y avoir qu'un seul élément correspondant à un id dans une page.
2. document.getElementsByTagName(*name*) Cette méthode va nous retourner la liste de tous les éléments d'une balise donné. document.getElementsByTagName(*a*) par exemple va nous retourner la liste de tous les liens <a> de la page.
3. document.getElementsByClassName(*name*) Cette méthode nous retourne la liste de tous les éléments possédant la valeur donnée dans l'attribut class.

Dans la script inscription.js (vide à ce moment), mettez le code suivant puis lancez votre page et regardez la console :

```
let mg = document.getElementById("menu-gauche")
console.log(mg)
```

Que remarquez-vous? L'élément recherché n'est pas là et il est plutôt écrit « null ». Pourtant, il y a bien un élément avec un id de valeur « menu-gauche » dans la page HTML, non? En fait, au moment où le script s'exécute, l'élément n'y est pas encore. Le script s'exécute dès que la balise <script> est lue et à ce moment là le reste de la page n'a pas encore été généré. Il faut donc attendre que ce soit le cas. Pour ce faire, il suffit d'attendre qu'un événement indiquant que la page est prête soit lancé. Il y a plusieurs événements possibles, chacun étant lancé à un moment légèrement différent selon si on veut que tous les scripts et images soient prêts ou alors seulement la page. Ici, seulement la page sera suffisante. Nous allons donc nous abonner à l'événement DOMContentLoaded. Cet événement se produit lorsque le DOM est prêt, mais pas nécessairement le reste.

De la même façon qu'il est possible de naviguer dans notre HTML à partir de Javascript, il sera possible de s'abonner à des événements.

```
document.addEventListener('DOMContentLoaded', (event) => {
    console.log(`DOMContentLoaded`);
});
```

La méthode addEventListener peut s'appeler sur tout objet représentant une balise. document représentant la racine de notre page, c'est un bon endroit pour s'inscrire aux abonnements plus « globaux ». Le premier paramètre est le nom de l'événement et le deuxième est la fonction à exécuter. Ici, nous utilisons la programmation fonctionnelle pour créer une fonction anonyme. (event) représente alors les paramètres de notre fonction sans nom et sa définition est donné par le contenu entre les accolades situées de l'autre côté du =>.

Si nous déplaçons le code écrit plus tôt dans cette fonction, nous avons maintenant accès à l'élément :

```
document.addEventListener('DOMContentLoaded', (event) => {
    console.log(`DOMContentLoaded`);
    let mg = document.getElementById("menu-gauche")
    console.log(mg)
});
```

La console affiche maintenant le résultat attendu.

addEventListener permet de souscrire à n'importe quel événement., y compris ceux que nous pourrions nous même créer. Le seul inconvénient est qu'on souscrit avec le nom de l'événement, plutôt qu'en s'ajoutant à une liste de souscrits. Ce n'est donc pas réellement une implémentation du patron de conception observateur. Les événements sont locaux à la balise. Il existe toutefois des mécanismes pour faire remonter les événements dans la hiérarchie de balises (voir le concept de bubble pour ceux que ça intéresse).

Nous allons maintenant attacher notre fonction de gestion du menu gauche à nos éléments <a> en utilisant cette fois ces nouveaux concepts qui nous permettent d'éviter de mettre du javascript dans notre HTML. Donc, dans notre DOMContentLoaded, nous allons récupérer nos liens et ajouter le listener pour le clic.

Nous avons donc ce code :

```
function menuGaucheClic(lien){
    console.log(lien)
}

function attacherListenerMenuGauche(){
    let menu = document.getElementById("menu-gauche");
    let liens = menu.children;
    for (let i = 0 ; i<liens.length ; i++){
        liens[i].onclick =  menuGaucheClic(liens[i]);
    }
}

document.addEventListener('DOMContentLoaded', (event) => {
    attacherListenerMenuGauche()
});
```

Regardez dans la console. Il y a un problème. Notre console.log s'est déjà exécuté, alors que normalement la fonction n'a pas été appelée. Ceci est causé par la présence du paramètre, qui provoque l'appel de la

fonction plutôt que son simple attachement en tant que variable. Ce code va donner le comportement souhaité :

```
function menuGaucheClic(lien){
    console.log(lien)
}

function attacherListenerMenuGauche(){
    let menu = document.getElementById("menu-gauche");
    let liens = menu.children;
    for (let i = 0 ; i<liens.length ; i++){
        liens[i].addEventListener('click', function(){
            menuGaucheClic(liens[i])
        });
    }
}

document.addEventListener('DOMContentLoaded', (event) => {
    attacherListenerMenuGauche()
});
```

Ce code présente aussi le fonctionnement régulier des boucles. C'est identique à Java, donc passons rapidement. Toutes listes javascript possède une propriété length qui donne le nombre d'élément. La propriété children d'un élément HTML (comme dans menu.children) donne accès aux éléments HTML imbriqués dans la balise courante.

Nous sommes donc revenus au même point que nous avions avant lorsqu'il y avait des attributs onclick dans notre html. Cette nouvelle méthode est un peu plus verbeuse, mais offre un bien meilleur découplage entre nos fichiers de code. Pour s'attaquer au prochain découplage, soit celui du CSS que nous avions dans notre Javascript, la solution sera un peu plus simple. L'idée sera simplement de venir ajouter une classe à notre élément et à laisser le CSS se faire tout seul. En effet, le navigateur surveille en tout temps l'ensemble des éléments pour détecter si des classes ou des id sont ajoutés ou retirer afin d'adapter le CSS calculé pour chaque élément. Si on ajouter une classe, la modification se fait donc toute seule.

Pour notre lien cliqué, nous allons ajouter le CSS suivant dans main.css :

```
#menu-gauche a.choisi{
    background-color: var(--fond-menu-choisi);
}
```

et la variable :

```
:root{
    --fond-menu : #444444;
    --couleur-menu : aliceblue;
    --fond-menu-inverse: salmon;
    --couleur-menu-inverse: #223;
    --fond-menu-choisi: #123456;
}
```

Pour que le comportement du hover reste le même, il faut aussi le demander :

```
#menu-gauche a:hover, #menu-gauche a.choisi:hover{
    background-color: var(--fond-menu-inverse);
    color: var(--couleur-menu-inverse);
}
```

Pour appliquer la classe, il s'agit de l'ajouter à la liste des classes de l'élément :

```
function menuGaucheClic(lien){
    lien.classList.add("choisi");
}
```

Notre code sépare maintenant bien l'ensemble des responsabilités dans les bons fichiers.

Tant qu'à être là, nous pourrions aussi faire en sorte que seul notre élément le plus récent obtienne la classe « choisi » pour éviter que tout notre menu se retrouve bleu. Il suffit de parcourir nos éléments pour retirer la classe avant de la mettre sur le bon.

```
function menuGaucheClic(lien){
    let menu = document.getElementById('menu-gauche');
    let liens = menu.children;
    for (let i = 0 ; i<liens.length ; i++){
        liens[i].classList.remove("choisi")
    }
    lien.classList.add("choisi");
}
```

classList.remove ne provoque pas d'erreur si on tente d'enlever une classe qui n'y est pas, donc nous n'avons pas besoin de nous en préoccuper. Vous remarquerez que j'ai maintenant quelques lignes qui se dupliquent entre ma fonction et mon listener. Nous pourrions les extraire dans une autre fonction, à votre plaisir de puriste.

C'est donc tout le contenu pour la page inscription, que vous remettrez ainsi.

5. Génération de contenu

Javascript ne sert pas seulement à réagir à des événements sur la page pour ajouter de l'interaction minimale. Il peut aussi servir à complètement générer sur contenu. Nous apprendrons cette technique dans la page des points de ventes.

Créez un script nommé points-de-vente.js et importez-le dans la page point_de_vente.html. C'est aussi le moment de télécharger le fichier « commerces.json » disponible sur la page Moodle du cours. Mettez ce fichier dans le projet, au même niveau que le reste. Ce fichier va contenir les données sur les points de vente que nous aurons dans notre page.

Allez maintenant dans le fichier « point_de_vente.html » et effacer la liste des <article> pour ne conserver que la <section> :

```
<section id="section-pdv">
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit. Adipisci
aliquid animi eligendi iure, molestias, necessitatibus nostrum numquam
placeat sapiente sit tempore totam vel voluptatem? Dicta explicabo libero
```

```
officiis quisquam repellat.</p>
    <h2 id="para-2">Points de ventes au Canada</h2>
    <section id="pdv-grille">
    </section>
    <h2 id="para-3">Points de ventes à l'international</h2>
    <section id="pdv-flex">
    </section>
</section>
```

Bien. L'objectif de cette section sera donc de générer le contenu de cette page à partir de la liste des commerces récupérées dans le fichier. Pour cette semaine, le fichier sera statique. Au prochain laboratoire, nous verrons comment charger cette liste via le serveur.

L'idée de la section est donc simple, son exécution plus complexe. Il s'agit de charger le fichier puis pour chaque élément de générer le <article> correspondant puis de l'insérer dans la page.

Au départ, nous avons donc ceci :

```
function genererMagasin(data){
    console.log(data)
}

document.addEventListener('DOMContentLoaded', (event) => {
    });
```

Lorsque la page est prête, nous allons vouloir appeler genererMagasin en lui passant la liste en paramètre. Pour récupérer des éléments distants en Javascript, nous allons utiliser la fonction fetch. Cette fonction retourne une promesse, c'est-à-dire un élément asynchrone. Nous l'utiliserons ainsi :

```
document.addEventListener('DOMContentLoaded', (event) => {
    fetch('./commerces.json')
        .then(commerces => {return commerces.json()})
        .then(data => genererMagasin(data) )
});
```

Le parameter de fetch correspond à l'adresse où nous voulons aller. Il est possible de charger n'importe quel type de fichier de cette façon. Le fetch prend un certain temps à se faire, il est donc fait en parallèle du reste de l'exécution. Ce qui vient dans le .then est alors les fonctions (anonymes et fonctionnelles) qui seront appliquées sur le résultat. D'abord, il faut convertir le résultat en json, avec la méthode .json. Puis on passe le json à notre fonction « generer_magasin ».

Le data reçu est de la forme d'un objet qui a deux attributs : « commerces_locaux » et « commerces_internationaux ». Chacun de ces attributs a pour valeur une liste d'objets représentant chacun un point de vente.

Commençons notre fonction générer en récupérant les endroits où nos commerces vont aller, ce que nous savons déjà faire :

```
function genererMagasin(data){
    let locaux = document.getElementById('pdv-grille');
```

```
    let internationaux = document.getElementById('pdv-flex');

    console.log(locaux);
    console.log(internationaux);
}
```

Puis, nous allons parcourir nos listes :

```
function genererMagasin(data){
    let locaux = document.getElementById('pdv-grille');
    let internationaux = document.getElementById('pdv-flex');

    for(let indice in data.commerces_locaux){
        let element = data.commerces_locaux[indice]
        console.log(element)
    }

    for(let indice in data.commerces_internationaux){
        let element = data.commerces_internationaux[indice]
        console.log(element)
    }
}
```

Notez ici l'équivalent du foreach en Javascript, avec la boucle for… in qui nous retourne quand même les indices. L'avantage est que nous n'avons plus à nous préoccuper des bornes.

Chacun des points de vente devrait maintenant s'afficher dans la console sous la forme d'un objet. La prochaine étape consiste donc à remplacer ces objets par un élément html. Il est possible de les générer sous la forme d'une chaîne de caractère contenant le html requis. Il est aussi possible de tout générer en créant directement les objets (similairement à ce qui a été fait en IFT287). Nous ferons les locaux en texte et les internationaux en code, pour voir la différence.

Voici le code par texte :

```
let innerHTML = ""
for(let indice in data.commerces_locaux){
    let element = data.commerces_locaux[indice]
    innerHTML += "<article class=\"pdv-item\">" +
            "<h1>" + element.ville + "</h1>" +
            "<address>"+ element.adresse + "</address>" +
            "<h2>Heures d'ouverture: </h2>" +
            "<ul>";
    for (let h in element.horaire){
        let jour = element.horaire[h];
        innerHTML += "<li>" + jour + "</li>";
    }

    innerHTML += "</ul></article>";

}

locaux.innerHTML = innerHTML;
```

On se déclare d'abord une variable pour tout contenir afin de modifier notre page qu'une seule fois par section. Puis il s'agit de recréer ce que nous avions avant. Cette méthode offre le désavantage de faire beaucoup de concaténation de chaîne, ce qui n'est pas optimal. Vous pourrez chercher les optimisations dans votre temps libre, si le sujet vous intéresse.

En dernière étape, on fait donc l'insertion de toute la chaine HTML en temps que innerHTML de notre balise.

Voici maintenant le code pour la génération avec les éléments :

```
for(let indice in data.commerces_internationaux){
    let element = data.commerces_internationaux[indice];
    let article = document.createElement('article');
    article.classList.add('pdv-item');

    let h1 = document.createElement('h1');
    h1.innerText = element.ville;
    article.appendChild(h1);

    let adresse = document.createElement('address');
    adresse.innerText = element.adresse;
    article.appendChild(adresse);

    let h2 = document.createElement('h2');
    h2.innerText = "Heures d'ouverture:";
    article.appendChild(h2);

    let ul = document.createElement('ul');
    for (let h in element.horaire){
        let jour = element.horaire[h];
        let li = document.createElement('li');
        li.innerText = jour;
        ul.appendChild(li);
    }
    article.appendChild(ul)

    internationaux.appendChild(article)
}
```

Pour chaque balise, nous utilisons « document.createElement » qui va nous créer un nœud correspondant à cette balise html. On lui ajoute une valeur textuelle avec « innerText » Lorsqu'on crée un nœud qui va à l'intérieur d'un autre, on l'ajoute avec la méthode « appendChild », qui va le mettre à la suite des enfants actuels. Lorsque le <acticle> est créé, on l'ajoute à la liste des enfants de notre section.

Pour ceux qui seraient intéresser à aller plus loin, HTML offre maintenant un concept de template qui permet de créer des modèles de code html qui ne s'affichent pas dans la page, mais qu'il est possible de dupliquer puis de remplir pour finalement les ajouter dans la page. C'est une solution théoriquement plus complexe, mais qui fourni un code plus lisible et un peu moins compliqué.

Voilà, c'est tout pour ce laboratoire. Vous devez remettre l'ensemble du projet, avec les pages index, inscriptions et points_de_vente modifiées.