# Introduction à JavaScript

Pierre Pompidor

11 novembre 2021

## Table des matières

# 1 Premier aperçu du langage JavaScript :

**JavaScript** est un langage de programmation créé en 1995 par *Brendan Eich* qui travaillait pour la société *Netscape*. Le but originel du langage est de créer des scripts (c'est à dire des programmes interprétés), qui sont exécutés par un navigateur, principalement pour manipuler les données du **DOM** (*Document Object Model*), c'est à dire les objets (au sens informatique du terme) représentant les éléments d'un document balisé (par exemple ceux d'une page HTML) alloués en mémoire du navigateur.

JavaScript a ensuite beaucoup évolué fonctionnellement (par exemple en pemettant l'accès asynchrone (non bloquant) à des données fournies par le serveur) et a même récemment investi "le côté serveur" avec l'environnement *Node.js*.

Malgré son nom qui peut porter à confusion, JavaScript a très peu de liens avec la langage *Java* (en fait seulement quelques structures syntaxiques qui proviennent en fait du langage *C*).

Le langage JavaScript se situe à la confluence de deux paradigmes de programmation : la paradigme de la **programmation fonctionnelle** et celui de la **programmation par objets**, et cela le rend difficile à maîtriser.

*Le but de ce cours n'est pas d'apprendre en profondeur ce langage, mais de montrer quelques fonctionnalités simples à mettre en œuvre.*

## 1.1 Où coder du code javascript et comment le déboguer ?

Pour une utilisation côté client (*client-side*) hors framework, les codes JavaScript doivent être externalisés dans des fichiers d'extension **.js** et liés aux codes HTML via la balise `<script>`.

Soit un fichier JavaScript nommé *bonjour.js* ne contenant que la ligne suivante : `console.log("Bonjour !");`

Et la page HTML *test.html* mettant en œuvre ce script :

```
<html>
   <head>
       <script src="bonjour.js" type="text/javascript" language="javascript">
       </script>
   </head>
   <body> JavaScript vous dit bonjour dans la console </body>
</html>
```

Cet exemple de code est exécutable en étant chargé (ouvert) dans votre navigateur.

Les messages générés par la méthode *log()* de l'objet *console* seront affichés dans la console qui fait partie des **outils du navigateur** (appuyez sur la touche `<F12>` pour la faire apparaître).

Remarque : surtout n'utilisez jamais une balise `<script>` auto-fermante !

## 1.2 Exemple d'un code JavaScript associé au CV de James qui zoome sa photo :

Une fonctionnalité très usuelle de JavaScript est la mise en place d'actions suite aux actions de l'internaute.

Différents **gestionnaires d'événements** peuvent être associés aux éléments HTML. Dès qu'un gestionnaire est mis en œ]uvre, il scrute les événements produits par les actions de l'internaute, ou le changement de l'état de la page web, correspondants à sa raison d'être.

Par exemple, quand l'internaute clique, ou survole avec le pointeur de souris, une image ou une chaîne de caractères, un événement est déclenché ce qui provoque l'exécution de la fonction JavaScript associée.

Voici les principaux gestionnaires d'événements qui correspondent à des attributs à ajouter aux balises HTML concernées :

— `onclick` : clic sur un élément
— `onmouseover` : entrée du pointeur de la souris sur un élément
— `onmouseout` : sortie du pointeur de la souris d'un élément
— `onselect` : sélection d'un contrôle (par exemple un item de liste déroulante)
— `onload` : chargement de la page dans le navigateur (cet attribut doit accompagner la balise `<body>`)
— `onunload` : l'internaute quitte la page (cet attribut doit accompagner la balise `<body>`)

Voici le schéma de programmation qui permet d'associer un gestionnaire d'événements qui traite les clics souris effectués sur une image :

```
<img src="<lien vers la photo>" onclick="<appel de la fonction JavaScript>" />
```

### 1.2.1 Mise en place de l'écouteur d'événements (survol de la souris) :

Notre but est de pouvoir agrandir à sa taille originelle la photo de James quand celle-ci est survolée par le pointeur de la souris. Voici le code mettant en œuvre les deux gestionnaires d'événements :

```
<!doctype html>
<html>
  <head>
     <meta charset="UTF8" />
     <title> CV de James Bond avec zoom </title>
     <link rel="stylesheet" href="styles_CV_James.css" />
     <script src="CV_James.js"></script>
  </head>

  <body>
     <header>
      <p> James Bond <img id="photo" src="Connery.jpg" onmouseover="zoom(this)"
                                                      onmouseout="dezoom(this)" /> </p>
        Nationalité britannique <br/>
        Profession : agent secret (dernier employeur : MI6)
     </header>
     ...
  </body>
</html>
```

Les deux gestionnaires/écouteurs d'événements `onmouseover` et `onmouseout` appellent respectivement une fonction JavaScript qui zoome la photo et une autre qui la remet dans sa taille réduite.

`this` représente l'élément du DOM sur lequel l'événement a porté (ici l'image) (techniquement cette variable contient l'adresse en mémoire de cet élément).

### 1.2.2 Fonctions JavaScript qui zoome et dézoome la photo :

Voici un aperçu du contenu du fichier *CV_James_avec_zoom_photo.js* qui contient le code JavaScript permettant d'agrandir la photo quand celle-ci est survolée par le pointeur de la souris :

```
var largeurImage;
var hauteurImage;

function zoom(image) {
    largeurImage = image.style.width;
    hauteurImage = image.style.height;
    image.style.width = "auto";
    image.style.height = "auto";
}

...
```

Ce code se décompose en trois parties :

Deux **variables** `largeurImage` et `hauteurImage` sont déclarées et valuées avec la largeur et la hauteur de l'image (définies dans la feuille de style) pour pouvoir remettre l'image dans sa dimension initiale quand le pointeur de la souris la quittera.

Le **typage des variables est dynamique** en JavaScript, c'est à dire que comme en Python, il n'est pas besoin de préciser un typage (entier, réel, caractère, chaîne de caractères, ...) lors de leurs déclarations.

Le mot réservé `var` permet de déclarer une nouvelle variable globalement (elle sera connue dans tout le code), à la différence de `let` qui restreint sa visibilité (sa "portée") au bloc d'instructions dans lequel elle apparaît.

La fonction `zoom()` qui est invoquée par le gestionnaire d'événements `onmouseover`.

Une fonction JavaScript est déclarée par le mot réservé `function`.

Les instructions assujetties à la fonction sont circonscrites par ses accolades (ouvrante et fermante) qui délimitent un **bloc d'instructions**.

Son paramètre correspond à la référence de l'image : il contient l'adresse en mémoire de l'objet (une zone mémoire implémentant une liste de duos clefs/valeurs) qui contient les différents attributs (ou propriétés) de l'image. Ainsi la largeur et la hauteur de l'image sont stockées comme valeurs des deux propriétés `width` et `height` contenues dans un sous-objet en valeur de la propriété `style` de l'objet `image`.

La notion d'`objet` liée au paradigme de la programmation par objets ne sera pas approfondie ici. Il faut juste signaler que contrairement aux autres langages objets majeurs (C++, Java, ...) la programmation par objets en JavaScript ne nécessite pas la définition de classes (même s'il est possible depuis sa dernière évoluation d'en utiliser).

Cette fonction recopie la largeur et la hauteur de l'image dans les variables globales `largeurImage` et `hauteurImage`, puis value les propriétés de l'image en mémoire du navigateur à `auto`, c'est à dire aux valeurs intrinsèques de l'image.

La fonction `dezoom()` qui est invoquée par le gestionnaire d'événements `onmouseout` (le pointeur de souris a quitté l'iamge). Elle remet les valeurs de largeur et de hauteur de l'image à ses valeurs initiales.

*Le choix des noms de variable est à la discrétion du programmeur, mais il est fortement recommandée que leur sémantique soit forte pour une bonne lisibilité du code, et qu'elles soient définies soient en français, soient en anglais, mais en évitant un mélange des deux.*

# 2 Rôle et syntaxe de JavaScript :

## 2.1 Rôle de JavaScript comme accesseur du DOM :

Le **DOM** (Document Object Model) représente toutes les données mises en mémoire du navigateur suite au chargement d'une page.

En effet, quand du code HTML est transmis au navigateur, celui-ci va utiliser un outillage spécifique pour allouer en mémoire de l'ordinateur les différentes entités de ce code (balises, attributs, informations textuelles, ...). Cet outillage est constitué de classes au sens de la programmation par objets - ce point ne sera pas détaillé dans ce support - et est nommé le Document Object Model (le *DOM*). Par extension le DOM désigne également toutes les données mises en mémoire et associées à une page affichée dans une fenêtre du navigateur.

L'allocation d'une balise HTML est appelé un **élément**.

### 2.1.1 Pourquoi accéder au DOM :

L'accès au DOM est inévitable si on veut modifier le contenu d'une page sans être obligé de demander au serveur de nous renvoyer une nouvelle page. C'est par exemple le cas vu précédemment, où nous voulions agrandir une image en la survolant.

JavaScript a été initialement conçu pour assurer cette fonctionnalité. Il possède donc des fonctions spécifiques pour sélectionner un ou plusieurs éléments du DOM :

- via le nom d'un type de balises (par exemple toutes les balises `<img>`) ;
- via un identifiant (exprimé par un attribut `id`).

### 2.1.2 Sélection d'éléments dans le DOM (API DOM vs jQuery) :

Pour sélectionner des éléments du DOM (c'est à dire des balises allouées en mémoire du navigateur), il y a deux possibilités :

- utiliser les fonctions JavaScript de l'API DOM "de base" (comme `getElementById()` ou `getElementsByTagName()`) ;
- utiliser une bibliothèque JavaScript de plus haut niveau qui offre une syntaxe plus concise : la bibliothèque la plus populaire est **jQuery**.

La bibliothèque jQuery (version 3.5.1 au moment de l'écriture de ce support de cours) peut être directement importée en rajoutant le lien suivant dans l'entête de la page HTML :

```
<script src="https://code.jquery.com/jquery-3.5.1.min.js"
        integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
        crossorigin="anonymous"></script>
```

L'objet jQuery (qui nous permettra de sélectionner des éléments du DOM, d'invoquer un serveur pour récupérer des ressources...) est alors désigné par `$`

Voici comment par exemple sélectionner l'élément correspondant à l'image identifiée par l'identifiant `photo`, pour afficher le nom du fichier image dans la console JavaScript :

Avec l'API DOM :

```
console.log(document.getElementById("photo").src);
```

Avec la bibliothèque jQuery (qui reprend la syntaxe des sélecteurs CSS) :

```
console.log($("#photo").attr("src"));
```

Dans ce dernier exemple, la méthode *attr()* appliquée à l'objet du DOM sélectionné permet d'accéder à la valeur d'un attribut de cet élément (el le pourrait être aussi utilisé avec un second paramètre pour donner une valeur à un attribut).

### 2.1.3 Suppression d'éléments dans le DOM :

Différentes possibilités sont offertes par l'API DOM de JavaScript ou la bibliothèque jQuery pour supprimer des éléments du DOM.

Dans le cadre des TP, nous utiliserons seulement la méthode *empty()* de jQuery pour supprimer les éléments affiliés à un élément parent, par exemple les items d'une liste (ici identifiée par *listeAVider*) :

```
$("#listeAVider").empty();
```

## 2.2 Présentation des éléments de base de la syntaxe de JavaScript :

### 2.2.1 Déclaration des variables :

Les variables sont **typées dynamiquement** (et non pas lors de leurs déclarations). Elles sont déclarées par :
- le mot réservé **var** : la portée de la variable est globale, mais si elle est déclarée dans une fonction, la variable n'est pas visible dans les fonctions incluses) : depuis la création du mot réservé *let*, ce typage est beaucoup moins employé ;
- le mot réservé **let** : la portée de la variable est alors le bloc d'instructions ;
- le mot réservé **const** : la variable n'est accessible qu'en lecture.

Javascript va typer (en interne) les variables suivant six types primitifs :
- un type **booléen** : **true** et **false** ;
- un type **nul** : **null** (qui représente une adresse en mémoire qui ne pointe vers rien) ;
- un type **indéfini** (en résultat de l'accès à une variable qui n'existe pas ou qui n'a pas de valeur) : **undefined** ; exemple de variable créée mais de valeur indéfinie : `var i;`
- un type pour les **nombres** qu'ils soient entiers ou réels : **number** ;
- un type pour les **chaînes de caractères** : **string**
  les chaînes de caractères peuvent être circonscrites par des simples quotes ou des doubles quotes ; nous prendrons le parti dans nos exemples, de généralement les encadrer par des doubles quotes.
- un type pour les **symboles** (ce type est introduit par ECMAScript 6 et ne sera pas développé ici).

et le type **Object** dans les autres cas de figures : nous voyons déjà que JavaScript est bien un langage à objets.

Voici quelques exemples de création de variables scalaires (en rangeant par simplicité dans cette catégorie les chaînes de caractères) :

```
var bizarre;  // sa valeur est : undefined
var i = 0;
let bool = true;
const nom = "Immuable";
```

### 2.2.2 Les structures de données usuelles :

En JavaScript, deux structures de données sont omniprésentes :
- les **objets** ;
- les **listes** (qui correspondent à des tableaux dynamiques et qui sont par ailleurs des objets).

*Nous présenterons d'abord les listes avant les objets.*

### 2.2.3 Les listes :

Les listes sont créées soit en utilisant la fonction constructrice *Array()* (les fonctions constructrices ne seront pas détaillées dans ce support), soit directement en employant les crochets (qui ne sont que du sucre syntaxique).

Par exemple, si nous voulons créer une liste nommée *maListe* qui contiendra les chiffres 1, 2 et 3 et la chaîne de caractères "partez", voici les deux façons de faire :

```
let maListe = new Array(1, 2, 3, "partez");
let maListe = [1, 2, 3, "partez"];
```

Le nombre d'éléments d'une liste est donné par l'attribut *length*

```
console.log(maListe.length);
```

Il est possible de tester l'existence d'une valeur dans une liste avec la méthode *includes()* (ici la chaîne "partez" dans la liste *maListe*) :

```
if (maListe.includes("partez")) { ... }
```

### 2.2.4 Les objets et les fonctions constructrices :

Contrairement à de nombreux autres langages de programmation, les objets peuvent être créés littéralement en JavaScript (et ne sont pas originellement des instances de classes). La syntaxe qui permet de formater ces objets lorsque ceux-ci sont sérialisés (càd écrits dans un fchier texte) est appelée **JSON** (*JavaScript Object Notation*). Cette syntaxe est devenue extrêmement populaire dans l'échange de données (en rendant XML beaucoup moins utilisé).

Pour information, la gestion des classes en JavaScript a été introduite via la norme ECMAScript 6 et des extensions "de plus haut niveau" comme **TypeScript** (`http://www.typescriptlang.org/`) ou **Dart** (`https://www.dartlang.org/`.

Dans cet exemple, un objet littéral *monObjet* est défini comme suit :

```
let monObjet = { "nom": "Bond",
                 "prénom": "James",
                 "acteurs": ["Sean Connery", "George Lazenby", "Roger Moore",
                             "Timothy Dalton", "Pierce Brosnan", "Daniel Craig"],
                 "films": {"1962" : "James Bond 007 contre Dr. No",
                           "1963" : "Bons baisers de Russie",
                           "1964" : "Goldfinger"}
               };
```

Voici quelques exemples de manipulation de cet objet (vous remarquerez que la valeur de la troisième propriété est une liste et celle de la quatrième un objet) :

```
console.log(monObjet.nom);         // Bond
console.log(monObjet.acteurs[0]); // Sean Connery
console.log(monObjet.films[1964]); // Goldfinger
```

Les objets peuvent être aussi instanciés par des **fonctions constructrices**.

Voici un exemple d'une fonction constructrice *personne()* qui crée un objet *bond*.

Le mot réservé **new** alloue une zone mémoire dans laquelle seront associées variables (ici *nom* et *prenom*) et méthodes (ici *cv()*) qui utilisent ces variables. Le mot réservé **this** représente dans la classe la référence à la zone mémoire dévolue à l'objet en cours de construction.

```
function personne(nom, prenom) {
   this.nom = nom;
   this.prenom = prenom;
   this.cv = function() { console.log("Je suis "+this.nom+" "+this.prenom); };
}

let bond = new personne("Bond", "James");
bond.cv();  // Je suis Bond James
```

### 2.2.5 Les blocs d'instructions et la structure conditionnelle :

En JavaScript les blocs d'instructions sont généralement circonscrits par des **accolades**. Dans le cas où une seule instruction est assujettie au bloc, les accolades peuvent être omises. Mais même dans ces cas-là, il est préférable de garder les accolades ce qui est une sécurité lors de l'ajout d'une nouvelle instruction au bloc.

Les structures conditionnelles et itératives créent des blocs d'instructions.

La structure conditionnelle présente le schéma suivant :

```
if ( <expression conditionnelle> ) {
   ...
}
else { // ce bloc est bien sûr facultatif
   ...
}
```

### 2.2.6 Les structures itératives

Différentes structures itératives existent en JavaScript :
— la structure **while (...) ...**
— la structure **for (... in ...) ...**
— la structure **for (... of ...) ...**
— la méthode **forEach()** s'appliquant aux listes.

Voici deux exemples de mise en œuvre des structures *for* (qui sont les plus usitées).

Structure *for (... in ...)* :

la boucle *for (... in ...)* permet d'accéder aux indices des éléments de la liste :

```
let voitures = ["Aston Martin", "Ford Mustang", ...];
for (let i in voitures) {
     console.log(i +" : "+voitures[i]); }
```

Structure *for (... of ...)* :

la boucle *for (... of ...)* permet d'accéder aux valeurs des éléments de la liste :

```
for (let v of voitures) { console.log(v); }
```

La méthode **forEach()** s'appliquant aux listes et proposée par la bibliothèque jQuery sera présentée dans le chapitre concernant AJAX.

# 3 Principe de l'importation de données d'un serveur (AJAX) :

L'acronyme **AJAX** (Asynchronous Javascript And XML) désigne une fonctionnalité maintenant ancienne de JavaScript qui permet de récupérer d'un serveur – de celui qui a délivré la page ou d'un serveur tiers - des **données** (et non du HTML) qui permettent de mettre à jour une page sans modifier sa morphologie (un exemple classique est de faire apparaître le nom d'une ville suite à la spécification par l'internaute de son adresse postale).

Le serveur qui délivre les données peut être de différents types, mais par souci d'homogénéité, nous allons présenter un exemple de serveur JavaScript **Node.js** créé grâce à un (mini-)framework nommé **Express**.

Les données importées étaient intialement formatées en **XML** (Extensible Mark-up Language), mais sont de plus en plus formatées en **JSON** (JavaScript Objet Notation) qui correspond à la sérialisation (c'est à dire la notation textuelle) d'objets JavaScript.

Enfin nous allons utiliser la bibliothèque JavaScript **jQuery** pour coder de manière plus plus concise.

## 3.1 AJAX côté serveur :

### 3.1.1 Le fichier de données JSON :

Dans notre exemple, les données doivent être créées dans un fichier texte.

Pour les formater, la structure la plus usuelle (mais pas la seule) est de créer une liste d'objets (*collection*) :

```
[{
  "<nom de propriété>" : <valeur de propriété>,
  "<nom de propriété>" : <valeur de propriété>,
  ...
 },
 ...
]
```

Comme nous l'avons vu, une liste est encadrée par des crochets, et un objet par des accolades.

Voici maintenant le fichier JSON *cocktails.json* qui liste quelques cocktails prisés par les différents James Bond :

```
[{"nom":"VM",
  "amateurs":["Sean Connery", "George Lazenby", "Roger Moore",
              "Timothy Dalton", "Pierce Brosnan", "Daniel Craig"]
 },
 {"nom":"Vesper", "amateurs":["Daniel Craig"]},
 {"nom":"Collins", "amateurs":["Sean Connery"]},
 {"nom":"Mint J.", "amateurs":["Sean Connery", "George Lazenby"]},
 {"nom":"The Macallan", "amateurs":["Craig"]}
]
```

Ce fichier décrit une liste (encadrée par des crochets) contenant des objets (encadrés par des accolades).

Il est à noter que ces données pourraient être stockées dans une base de données NoSQL comme *MongoDB*

### 3.1.2 Le serveur Node.js :

Un **serveur Node.js** est un programme JavaScript exécuté par un interpréteur JavaScript particulier nommé Node.js côté serveur.

Ce programme JavaScript "tourne" en permanence, attend d'être invoqué via une requête HTTP, et si c'est le cas, vérifie si le message (de la **route** portée par la requête HTTP correspond à un de ses **services web**. Dans l'affirmative, il exécute le code correspondant dont la finalité est de renvoyer des données au client (celui qui a émit la requête HTTP).

Les données renvoyées par le serveur sont généralement formatées en JSON (notation textuelle d'objets JavaScript) et proviennent d'une extraction d'une base de données (mais cela ne sera pas le cas dans notre exemple).

Un (mini) framework nommé **express** permet de créer les services web qui définissent les réponses à donner à différents messages (appelés des routes).

Voici le code de notre serveur (qui va "écouter" sur le port 8888) :

```
var express = require("express");
var app = express();
app.listen(8888);

app.get('/', function(request, response) {
    response.sendFile('CV_James_AJAX.html', {root: __dirname});
});

app.get('/fichier/:nomFichier', function(request, response) {
    console.log("renvoi de "+request.params.nomFichier);
    response.sendFile(request.params.nomFichier, {root: __dirname});
});

app.get('/cocktails', function(request, response) {
    console.log("renvoi de cocktails.json");
    response.sendFile('cocktails.json', {root: __dirname});
});
```

Par rapport à ce code, il faut donc noter que le serveur gère trois **services web** :

— le premier renvoie la page HTML sur la route `/` : le CV sera donc chargé par le navigateur via l'URL : `http://localhost:8888`

— le second renvoie les codes JS et CSS ainsi que les images sur la route `/fichier/<nomFichier>`
la route contient donc un paramètre qui est accessible via l'objet **params** de l'objet **request**

— le troisième renvoie les données JSON sur la route `/cocktails`

Les deux premiers services web sont dus à un impératif de sécurité : si des données additives sont renvoyées par un serveur (ce qui est le cas ici avec les cocktails), il faut par homogénéité que **toutes** les autres ressources du site à destination du navigateur (HTML, CSS, JS, images...) soient aussi renvoyées par le même serveur, autrement une alerte **CORS** (*Cross Origin Ressource Sharing*) serait levée par le navigateur et l'importation bloquée. (D'autres solutions techniques existent mais sont tout aussi complexes que de mettre en place celle présentée précédemment.)

la méthode *sendFile()* de l'objet *response* renvoie directement au navigateur le fichier en positionnant automatiquement les entêtes des trames HTTP (le navigateur sachant donc le type - HTML, image, JSON... - de la ressource reçue) .

Le second paramètre `{root: __dirname}` indique que le fichier est localisé dans le dossier courant.

Le serveur pourrait aussi (via la méthode *readFileSync()* de l'objet fs accéder au contenu d'un fichier pour appliquer des traitements métier aux données avant de les renvoyer.

Dans la cas d'un fichier au format JSON contenant des objets JavaScript serialisés ("textualisés"), ces données devraient donc être désérialisées (reconstruites en mémoire) avec la méthode *JSON.parse()* avant d'être modifiées, puis resérialisées avec la méthode *JSON.stringify()*.

Voici un fragment de code qui (dans un service web) instancie en mémoire des objets JavaScript contenus dans un fichier au format JSON pour ensuite les renvoyer modifiées ;

```
let texte = fs.readFileSync('<nomDuFichier>.json','utf8');
let data = JSON.parse(texte);
// des traitement métier sous appliqués aux données
response.end(JSON.stringify(data));
```

L'objet *fs* qui permet d'interagir avec le système de fichiers doit être préalablement créé :

```
var fs = require('fs');
```

## 3.2 AJAX côté client :

### 3.2.1 Code jQuery d'importation des données et leur encapsulation dans du HTML :

Nous allons utiliser la bibliothèque *jQuery* pour invoquer le serveur et récupérer de manière asynchrone les données que celui-ci renvoie. Nous pourrions aussi utiliser l'objet JavaScript *XMLHttpRequest* mais le code serait plus long...

Voici la structure d'une fonction JavaScript qui appelée lors de la sélection du lien :

— importe un fichier JSON qui contient une liste d'objets en envoyant une requête à un serveur ;

— parcourt la liste pour accéder à chaque objet ;

— inclut la valeur d'une propriété de l'objet courant pour l'intégrer dans une chaîne de caractères HTML :

— attache cette chaîne de caractère à un élément du DOM.

```
function afficherCocktails() {
   $.getJSON("<URL serveur>", function(data) {
       let html = "";
       $.each(data, function(index, objet) {
           html += "<création d'un fragment HTML>";
       });
       $("<point d'attachement du nouveau code HTML>").append(html);
   });
}
```

`$` désigne l'objet de plus niveau de la bibliothèque jQuery : à cet objet sont attachées des fonctions (appelées, dans le contexte de la programmation par objets, méthodes).

`$.getJSON()` est la méthode qui permet de télécharger un fichier JSON et d'allouer en mémoire la liste qui y est décrite (variable `data`).

`$.each()` est la méthode qui permet de parcourir une liste et en extraire chaque objet (variable `objet`, la variable *index* qui numérote chaque objet à partir de 0 n'est pas utilisée dans cet exemple).

Les méthodes de la bibliothèques *jQuery* font couramment usage de fonctions de rappel (ou fonctions de callback). Ces fonctions sont des fonctions anonymes passées en paramètres à des fonctions qui les appellent en retour pour renvoyer des données (souvent de manière asynchrones).

Voici donc la fonction JavaScript qui utilise du code jQuery pour importer le fichier de données, extraire de chaque objet le nom du cocktail, et l'intégrer dans du code HTML :

```
function afficherCocktails() {
   $.getJSON("http://localhost:8888/cocktails", function(data) {  // cocktails.json est renvoye
                                 // par le serveur (qui "écoute" sur le port 8888) sur la route /cocktails
       let html = "";
       $.each(data, function(index, objet) { html += "<li>"+objet.nom+"</li>"; });
       $("#listeCocktails").append(html);
   });
}
```

La chaîne de caractères `html` aura finalement la valeur suivante :

```
<li>VM</li><li>Vesper</li><li>Collins</li><li>Mint J.</li>
```

jQuery fait usage de fonctions de rappel (fonctions anonymes passées en paramètres). Nous en avons vu deux exemples étant celle de la méthode `$.getJSON()` qui récupère de manière asynchrone les données du serveur et celle de la méthode `$.each()` qui permet de parcourir les éléments d'une collection.

### 3.2.2 Digression sur les fonctions anonymes passées en paramètres (les fonctions de rappel) :

L'exemple précédent met en exergue que JavaScript - et ici JQuery - fait usage de **fonctions de rappel** (ou **callbacks**) qui sont des fonctions anonymes (*lambda fonctions*) passées en paramètres à une fonction.

Pour illustrer ce point, voici ci-après deux exemples de l'implémentation d'une fonction/méthode `myForEach()` qui :
— soit prend en paramètre une liste ;
— soit est associée à une liste considérée comme étant un objet (la fonction est donc une méthode).
et prend en paramètre une fonction de rappel qui sera exécutée autant de fois que cette liste comprend d'éléments (en affichant l'indice et la valeur de chaque élément de la liste).

Première illustration de la mise en œuvre d'une fonction de rappel :

```
let liste = [1, 2, 3, "partez !"];

let myForEach = function(liste, callback) {
    for (let i in liste) callback(i, liste[i]);
}

myForEach([1, 2, 3, "partez !"],
          function(i, element) {
             console.log(i, ":", element);
          });
```

Seconde illustration de la mise en œuvre d'une fonction de rappel :

```
let liste = new Array(1, 2, 3, "partez !");

Array.prototype.myForEach = function(callback) { for (let i in this) callback(i, this[i]); }

liste.myForEach(function(i, element) { console.log(i, ":", element); });
```

La fonction constructrice *Array* permet de créer des objets "liste" : la programmation objets et l'utilisation de fonctions constructrices ont été sommairement abordées précédemment.

L'objet *prototype* associé à la fonction constructrice *Array()* permet de mettre en place un mécanisme d'"héritage" entre objets qui ne sera pas abordé ici.

L'utilisation d'une fonction de rappel pour parcourir les éléments d'une liste lors de la mise en œuvre des méthodes `forEach()` de javaScript ou `$.each()` de jQuery est un style de programmation. **En revanche, celle mobilisée dans la méthode *getJSON()* est cohérente avec l'asynchronisme de la récupération des données, cette fonction de rappel n'étant activée qu'au moment où les données seront effectivement renvoyées par le serveur.**

### 3.2.3 Code HTML qui via jQuery charge des données complémentaires au CV de James :

Modifions le code HTML du CV de James pour :

- insérer un lien hypertexte dans la rubrique "hobbies" qui va permettre d'invoquer la fonction JavaScript *afficherCocktails()* d'importation des données :
- créer une balise `<ul>` identifiée par un identifiant qui nous permettra d'accrocher le code HTML formé à partir des données importées ;
- et sans oublier de créer un lien sur la bibliothèque *jQuery* dans le head de la page HTML.
  Cette bibliothèque peut être téléchargée du site jQuery ou comme ici, directement importée.

Voici les fragments de codes ainsi modifiés (importation de la bibliothèque jQuery et création d'une liste identifiée par ) :

```
<!doctype html>
<html>
  <head>
     ...
     <script src="https://code.jquery.com/jquery-3.5.1.min.js"
             integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
             crossorigin="anonymous"></script>
  </head>

  <body>
     <header> ... </header>
     <main>
        ...
        <article>
           <p> Hobbies : </p>
           <ul>
              <li> <a onclick="afficherCocktails()" href="#">Dégustation de nombreux cocktails</a>
                   <ul id="listeCocktails"></ul>
              </li>
              <li> Conversation mondaine en présence de public féminin </li>
           <ul>
        </article>
     </main>
  </body>
</html>
```

# 4 Mise en œuvre de la bibliothèque OpenLayers :

La bibliothèque **OpenLayers** qui utilise **OpenStreetMap** comme fournisseur de "tuiles" cartographiques permet l'affichage de cartes et la gestion d'**overlays** qui sont des surcouches à celles-ci (pour afficher des marqueurs, des informations textuelles...). Cette bibliothèque est libre d'utilisation (contrairement à **Google Maps**).

## 4.1 Affichage d'une carte centrée sur Montpellier :

Le code suivant permet d'afficher une carte centrée sur Montpellier (à un niveau de zoom arbitraire) :

```
var map = new ol.Map({
    target: 'map',
    layers: [new ol.layer.Tile({source: new ol.source.OSM()})],
    view: new ol.View({
          center: ol.proj.fromLonLat([3.876716,43.61]),
          zoom: 14
    })
});
```

Vous remarquerez avec un certain enthousiasme que :
— la bien nommée classe **Map** crée une carte ;
— *target* spécifie la division d'accueil dans la page HTML ;
— le fournisseur de tuiles ("tiles") est bien *OpenStreetMap*.

## 4.2 Affichage sur la carte de marqueurs correspondant à des points d'intérêt :

Si nous voulons créer des marqueurs correspondants à des points d'intérêts (ceux-ci pouvant par exemple être dans le cadre du TP associé à cette partie du cours, des restaurants, des bars...), *OpenLayers* nous permet la création d'**overlays** correspondant à des surcouches sur la carte et implémentées par des objets instances de la classe **Overlay**. Et à chaque overlay sera associée une image qui sera un clone de l'image "prototype" dont nous disposons (par exemple celle du célébre marqueur rouge).

Nous devons donc créer une liste qui contiendra nos marqueurs (en fait nos *objets overlays* et pour chaque point d'intérêt :
— cloner l'image prototype et l'insérer dans le body ;
— créer un objet overlay et lier l'image à celui-ci ;
— associer l'overlay à la carte.

Voici les éléments techniques dont nous avons besoin :

pour cloner l'image prototype, la méthode **clone()** :

```
let image = $("#markerProto").clone();
```

pour créer un *objet overlay* et lier l'image à celui-ci :

```
new ol.Overlay({position: ol.proj.fromLonLat([pi.long, pi.lat]),
                positioning: 'center-center',
                element: document.getElementById("<id>")});  // element fait référence à l'image
```

pour associer l'overlay à la carte, utilisez la méthode **addOverlay()** :

```
map.addOverlay(...);
```

Par ailleurs n'oubliez pas de forcer l'affichage des marqueurs si l'image prototype a été créée désaffichée (attribut **display** de son style à **none**) : `<img id="markerProto" style="display: none;" ... />`

et donc il faudra remettre le **display** de l'élément du DOM correspondant au marqueur à **block** (nous considérons ici que les marqueurs ont été stockées dans une liste nommée *markers*) :

```
markers[...].getElement().style.display='block';
```

## 4.3 Affichage d'une "popup" lors de la sélection d'un marqueur :

La programmation de cette tâche est similaire à la précédente. Les popups (par exemple des divisions affichant les noms des points d'intérêt) seront créées via des overlays et clonées à partir d'une division initiale spécifiée dans le HTML.

# 5 Composants graphiques (widgets) avec la bibliothèque jQuery-ui :

Si nous avons utilisé la bibliothèque *jQuery* pour sélectionner des éléments du DOM ou importer des données au format JSON, la bibliothèque **jQuery-ui** permet elle de créer des composants graphiques appelés **widgets**. Dans ce qui suit, nous introduirons seulement l'usage d'un **accordéon** (*accordion*) qui permet de regrouper différents éléments dans des "soufflets" (un seul soufflet étant à un moment donné ouvert).

## 5.1 Regroupement de cases à cocher dans une widget accordion :

Imaginons qu'ayant récupéré des données concernant des établissements géolocalisés (restaurants...) d'un serveur, nous voulons créer des cases à cocher, chacune d'entre elles correspondant à un de ces points d'intérêt que nous voulons faire afficher sur la carte par l'intermédiaire d'un marqueur.

Nous décidons que ces cases à cocher doivent être regroupées suivant le type de l'établissement (restaurant, bar...) auxquelles elles correspondent, et qu'étant très nombreuses, seules celles du type sélectionné par l'utilisateur doivent apparaître. Nous décidons alors de mettre en place une **widget accordéon** (un seul "soufflet" sera donc à un moment donné ouvert, les autres restant fermés).

Les cases à cocher doivent être définies dans des divisions elles-mêmes créées dans une division de plus haut niveau sur laquelle sera appliquée une méthode qui définira le type de widget voulue. La mise en œuvre d'un accordéon nous permet d'illustrer d'une manière plus générale l'utilisation de composants graphiques de la bibliothèque **jQuery-ui** (en sachant qu'il y a aussi pléthore d'autres bibliothèques de composants graphiques).

Des contraintes de "fabrication" sont requises pour la génération ultérieure d'un tel accordéon :
— les noms des soufflets doivent apparaître dans une balise `<h3>` avant chaque série de cases à cocher ;
— les séries de cases à cocher doivent être insérées dans une division qui suit la balise précédente.

Voici un exemple de code HTML **tel qui devrait être créé si nous le codions directement** pour pouvoir (dans le cadre du TP associé à ce cours) sélectionner des restaurants et des bars :

```
<div id="points_interet">
     <h3> restaurant </h3>
     <div id='restaurant'>
          <input type='checkbox' name='1'> restaurant1 </input>
          <input type='checkbox' name='2'> restaurant2 </input>
          ...
     </div>
     <h3> bar </h3>
     <div id='bar'>
          <input type='checkbox' name='...'> bar1 </input>
          <input type='checkbox' name='...'> bar2 </input>
          ...
     </div>
</div>
```

L'attribut *name* accompagnant les cases à cocher nous permettra de les dustinguer quand elles seront sélectionnées (via un numéro entier).

**Ce code devrait donc être produit dynamiquement suivant les données reçues du serveur.**

## 5.2 La création effective de l'accordéon :

Les cases à cocher doivent donc être gérées par un **accordéon** qui est une widget (composant graphique) de la bibliothèque **jQuery-ui**. Pour cela appliquez la méthode **accordion()** à la division *points_interet* comme suit :

```
$('#points_interet').accordion({collapsible: true, heightStyle: 'content'});
```

*Mais attention l'asynchronisme de JavaScript peut rendre cette instruction inopérante si celle-ci est appliquée avant que le DOM du navigateur ne soit mis à jour (càd avant que toutes les cases à cocher n'aient été créées)....*

# Index