**Introduction à JavaScript**
**-**
**A la rencontre de la programmation objets et fonctionnelle en JavaScript**
**Gestion des événements**
**Accès au DOM (API DOM de JavaScript et JQuery)**
**Chargement asynchrone de données (AJAX)**
**Interfaçage de (Google|OpenStreet) Maps**
**Création d'interfaces graphiques avec D3.js (et SVG)**
**Gestion de web sockets avec Node.js**
**Responsive web avec Bootstrap**

Pierre Pompidor

# Table des matières

# 1 Introduction à JavaScript

## 1.1 Un bref historique

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 une page HTML) alloués en mémoire du navigateur.

JavaScript a ensuite beaucoup évolué fonctionnellement (par exemple en pemettant l'accès asynchrone 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 en fait un langage assez difficile à maîtriser. Nous détaillerons ces deux facettes dans deux sections de ce chapitre.

## 1.2 Panorama de l'utilisation de JavaScript

Voici un aperçu non exhaustif de l'utilisation de JavaScript du côté client et du côté serveur.

Du côté client, les scripts JavaScript mis en œuvre par le navigateur sont utilisés :

— pour gérer l'événementiel lié à une page HTML, comme par exemple pour contrôler le contenu d'un formulaire avant sa soumission ou faire apparaître des popups ;

— pour accéder aux éléments du **DOM** (*Document Object Model*) et le cas échéant, les modifier. Le DOM représente la hiérarchie des objets créés par les balises (HTML...) en mémoire du navigateur. Cet accès peut être mis en œuvre l'API DOM de JavaScript, ou avec l'utilisation de bibliothèques de plus haut niveau comme *JQuery* ;

— pour gérer des flux de données asynchrones avec le serveur via l'architecture **AJAX** (*Asynchronous JavaScript and XML)* (souvent avec l'utilisation de la bibliothèque *JQuery*)

— pour gérer une communication bidirectionnelle et full-duplex entre clients et serveur(s) grâce aux **web sockets** (fréquente dans le cas de la programmation de jeux, de messageries instantanées...).

— pour créer des interfaces graphiques en utilisant la balise `<canvas>`, ou via les bibliothèques **D3.js** ou **dc.js** qui encapsulent **le langage SVG** (*Scalable Vector Graphics*) ;

— pour mettre en oeuvre un stockage de données sur le système de fichiers local via les objets *sessionStorage* et *localStorage*.

Les codes JavaScript mis en œuvre du côté serveur sont également utilisés pour gérer des web sockets, mais surtout pour créer des serveurs HTTP très réactifs car fondés sur l'architecture événementielle de JavaScript. Le principal environnement JavaScript permettant de créer de tels serveurs est **Node.js**.

*Node.js (immergé dans une architecture logicielle MEAN au côté du framework Angular) est enseigné dans l'inquiétante UE "Données du web" de première année de Master.*

## 1.3 Les bibliothèques et les frameworks applicatifs JavaScript

Plusieurs centaines de bibliothèques "utilitaires" et des dizaines de frameworks applicatifs JavaScript ont vu le jour depuis la création de ce langage. Il serait vain d'essayer d'en faire le tour dans cet ouvrage. Nous ne citons dans ce paragraphe que quelques références très populaires.

Voici cinq bibliothèques "utilitaires" qui sont sans conteste parmi les plus usitées :

— *JQuery* permet d'accéder et de modifier le DOM de manière très aisée, d'importer du serveur de manière asynchrone des données (via l'architecture *AJAX*) et propose un bel ensemble de widgets graphiques (par exemple pour créer des onglets, dérouler des accordéons, gérer le copié-collé (*drag and drop*) ;

— *D3.js* et *dc.js* permettent de créer des interfaces graphiques en 2D en encapsulant le langage *SVG* : la bibliothèque D3.js est introduite dans ce document tandis que la bibliothèque dc.js est quelque fois abordée dans l'UE "Données du web" ;

— *Three.js* permet de créer des scènes 3D ;

— *Bootstrap* permet quant à elle de gérer le *responsive web* (adaptation des interfaces aux différents écrans).

En ce qui concerne les frameworks applicatifs, les trois suivants sont actuellement les plus complets : **Angular**, *Vue.js* et *React* (qui est plus proche d'une bibliothèque de composants graphiques que d'un framework). Tous ces frameworks possèdent une communauté importante de développeurs et sont à la base de très nombreuses applications web (et peuvent être couplés côté serveur avec des environnements PHP, Java ou JavaScript). De ces trois, le framework Angular est celui qui offre indiscutablement le plus de modularité et permettra une production industrielle de code (mais c'est aussi celui qui est le plus complexe...).

# 2 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>`, comme dans l'exemple ci-après.

Soit notre script JavaScript nommé *bonjour.js* :

```
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 les consoles qui font parties des **outils du navigateur** (appuyez sur la touche `<F12>` pour les faire apparaître comme le paragraphe suivant l'expliquera). La méthode *console.dir(...)* permet quant à elle d'afficher le contenu d'un objet.

Remarque : il est très important de ne pas utiliser une balise `<script>` auto-fermante.

Si vous créez des codes JavaScript avec un framework (par exemple Angular), les règles de création vous seront données par celui-ci.

Du côté serveur (*server-side*), les codes JavaScript seront codés dans les dossiers associés à vos applications.

Les outils du navigateurs (appuyez sur la touche `<F12>` pour en faire apparaître un certain nombre) vous permettent :
— grâce aux *consoles*, de visualiser les messages émis :
  — par l'interpréteur JavaScript (notamment les erreurs de programmation (des liens cliquables vous permettent de vous positionner sur les lignes de code fautives) ;
  — par vos codes (comme dans l'exemple précédent) ;
— grâce aux onglets nommés *Eléments, HTML...*, d'afficher la hiérarchie des balises de la page web ;
— grâce aux onglets nommés *Inspecteur DOM...*, d'afficher la hiérarchie des objets alloués en mémoire du navigateur et correspondants à tous les documents chargés dans tous les onglets du navigateur.

Certains navigateurs proposent deux consoles :
— la console web qui affiche les messages (émis par les scripts JavaScript ou relatifs aux chargements des ressources induites) concernant la ressource chargée dans l'onglet courant ;
— la console du navigateur qui affiche les messages issus des ressources chargées dans tous les onglets.

# 3 Eléments de programmation de base

La syntaxe de base de la programmation en Javascript est celle du *langage C* elle-même reprise par le *langage Java*, mais cette syntaxe est dorénavant standardisée par **ECMAscript** qui définit un ensemble de normes de programmation.

## 3.1 Les 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 alors la fonction dans laquelle la variable apparaît (mais pas les fonction incluses) :
depuis la création du mot réservé *let*, ce typage est fortement déconseillé ;
— 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.

### 3.1.1 Les types internes

Javascript va typer (en interne) les variables suivant six types primitifs :
— un type **booléen** : **true** et **false** ;
— un type **nul** : **null** ;
— 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 (les fonctions en JavaScript sont aussi des objets) : 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 = "Pierre";
```

Le type d'une variable peut être testé par la fonction **typeof()**.

### 3.1.2 Le transtypage en JavaScript

Différentes fonctions et opérateurs permettent de transtyper une valeur. Les plus utiles en JavaScript sont celles qui transforment une chaîne de caractères (*string*) en valeur numérique (*number*) :
— `parseInt(<string>)` extrait (si elle existe) la valeur entière en préfixe de la chaîne :
`parseInt("123.45")` → 123
`parseInt("123.45bla")` → 123
— `parseFloat(<string>)` extrait (si elle existe) la valeur flottante en préfixe de la chaîne :
`parseFloat("123.45")` → 123.45
`parseFloat("123.45bla")` → 123.45
— l'opérateur unaire `+` extrait la valeur numérique que doit représenter la chaîne :
`+"123.45"` → 123.45
`+"123.45bla"` → NaN

Par ailleurs, l'opérateur de concaténation de chaînes de caractères est le signe arithmétique plus.
Voici un exemple de mise en œuvre :

```
let trois = 3;
console.log("1 2 "+trois+" partez !");
```

## 3.2 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).

Si les objets vont être longuement décrits dans la section "La programmation objets avec JavaScript", nous allons dire quelques mots sur les listes dans ce paragraphe.

Les listes sont créées soit en utilisant la fonction constructrice *Array()* (les fonctions constructrices sont présentées un peu plus loin dans cet ouvrage), 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);
```

Les méthodes suivantes (parmi beaucoup d'autres, mais celles-ci sont vraiment très utiles) peuvent être appliquées aux listes :
— *indexOf()* pour connaître la position de la première occurrence d'un élément dans la liste (renvoi -1 si l'élément n'est pas trouvé) ;
— *push()* pour rajouter un élément à la fin de la liste ;
— *splice()* pour ajouter ou supprimer des éléments à une position donnée dans une liste :
`<liste>.splice(<position>, <nombre d'éléments à supprimer>, [éléments à ajouter])`
par exemple :
  — ajout de la valeur *4* à la position *3* dans *maListe* : `maListe.splice(3, 0, 4)`
  — suppression de la valeur *4* qui vient d'être ajoutée : `maListe.splice(3, 1)`
— *sort()* pour trier les éléments de la liste.

Pour appliquer un traitement sur chaque élément d'une liste, il y a deux méthodes principales :
— *forEach()* qui permet d'appliquer un traitement sur chaque élément ;
— *map()* qui fait la même chose mais renvoie ensuite la liste modifiée.

Ces deux méthodes utilisent des fonctions de callback et leurs usages seront illustrées dans le paragraphe "Une fonction passée en paramètre (fonction de callback)".

## 3.3 L'application d'expressions régulières

Il est possible d'appliquer des expressions régulières sur des chaînes de caractères en JavaScript.
Une syntaxe légère permet d'encadrer l'expression régulière par des slashs (comme dans le mythique langage Perl) et d'appliquer à cette expression une méthode à laquelle la chaîne de caractères est passée en paramètre :
— la méthode **test()** pour tester si la chaîne de caractères comprend le motif de l'expression régulière ;
— la méthode **exec()** pour extraire des données de la chaîne.

Voici les deux syntaxes utilisant *test()* ou *exec()* :

```
/<expression régulière>/.test(<chaîne de caractères>)
/<expression régulière>/.exec(<chaîne de caractères>)
```

Voici un exemple où nous testons si la chaîne a le format (certes un peu imparfait) d'un email :

```
let mail = "pompidor@lirmm.fr";
if (/[\w.]+@[\w.]+\.\w+/.test(mail)) console.log("Format d'email");
```

Voici un exemple où nous extrayons le premier et le dernier nombre de la chaîne :

```
let string = "attention 1 2 3 partez !";
let data = /(\d+).+(\d+)/.exec(string);
if (data.length == 3) console.log(data[1]+" "+data[2]);
```

Dans ce dernier exemple, *data[0]* contient la partie de la chaîne de caractères qui a été couverte par l'expression régulière (ici *1 2 3*).

Dans le cas où nous devons interpoler la valeur d'une variable dans l'expression régulière, nous ne pouvons plus utiliser la syntaxe "légère" avec les slashs. Il faut créer l'expression régulière avec la fonction constructrice **RegExp()**.
Voici un exemple, qui affiche la châine de caractères comprise entre les valeurs des deux variables

```
let string = "attention 1 2 3 partez !";
let start = "attention";
let end = "partez";
let regexp = new RegExp(start+"(.+)"+end);
let data = regexp.exec(string);
if (data.length == 2) console.log(data[1]);
```

## 3.4 Les blocs d'instructions

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.

## 3.5 Les structures conditionnelles

Nous retrouvons en JavaScript les structures conditionnelles de base héritées du langage C.

### 3.5.1 La structure *if ... else ...* :

La structure conditionnelle de base est **if ... else ...**.

```
if ( stock == 0 ) {
   console.log("Le produit n'est plus disponible");
}
```

Le bloc "sinon" est introduit par l'instruction **else** :

```
if ( stock == 0 ) {
   console.log("Le produit n'est plus disponible");
}
else {
   console.log("Régalez-vous, achetez tout !");
}
```

L'enchaînement de conditions n'est pas introduite par une instruction particulière :

```
if ( stock == 0 ) {
   console.log("Le produit n'est plus disponible");
}
else if ( stock < 5 ) {
   console.log("Attention, il ne reste que quelques produits en stock");
}
else {
   console.log("Régalez-vous, achetez tout !");
}
```

### 3.5.2 La structure *switch* d'aiguillage multiple

L'aiguillage classique en programmation existe aussi en JavaScript.
En voici un exemple (incomplet) :

```
switch (codeErreur) {
  case 401 :
      console.log("utilisateur non authentifié");
      break;
   case 403:
      console.log("accès refusé");
      break;
   ...
   default :
      console.log("Code non identifié");
}
```

## 3.6 Les structures itératives

### 3.6.1 Les structures itératives avec indices de boucles

Les trois structures itératives classiques **while (...) ...**, **do ... while (...)** et **for (... ; ... ; ...)** existent en JavaScript. Pour illustrer rapidement ces structures itératives, implémentons l'affichage de la valeur d'une variable initialisée à 0 et incrémentée tant que sa valeur est inférieure à 10 :

**La structure while** La structure itérative de base fondée sur un "tant que" est bien entendu présente en JavaScript. En voici un exemple :

```
var i=0;
while (i < 10) {
    console.log(i); i++;
}
```

**La structure do ... while (...)**
Le contrôle de la boucle peut être inversé par la structure "faire ... tant que ..." :

```
var i = 0;
do {
    console.log(i);
    i++;
} while (i < 10);
```

**La structure for (... ; ... ; ...)**
C'est la structure classique permettant de manipuler l'indice de boucle dans l'entête de boucle existe aussi :

```
for (let i=0; i < 10; i++) {
    console.log(i);
    i++;
}
```

### 3.6.2 Les structures itératives sans indices de boucles

Deux structures itératives "automatiques", c'est à dire n'utilisant pas d'indice de boucle, peuvent être mises en œvre :
— la structure **for (... in ...) ...**
— la structure **for (... of ...) ...**
Il faut aussi noter la méthode **forEach()** s'appliquant aux listes et dont un exemple sera présenté dans le paragraphe traitant des fonctions de callback.

**La structure for (... in ...)**
La structure **for (... in ...) ...** permet aussi bien d'itérer sur tous les éléments d'une liste (via leurs indices) que sur les propriétés d'un objet (la notation "littérale" des objets sera expliquée dans la section suivante).

Voici un premier exemple de l'utilisation de cette structure itérative qui permet ici d'accéder aux indices des éléments d'une liste (qui contient des noms de marques mystérieuses) :

```
let brands = ["Konia", "Peach", "Threestars"];
for (let i in brands) {
     console.log(i +" : "+brands[i]);  // 0 : Konia
                                       // 1 : Peach
                                       // 2 : Threestars
}
```

Et voici un autre exemple sur l'accès aux propriétés d'un objet (la notion d'objet est détaillée dans une section ultérieure) :

```
let product = {"type": "phone", "brand": "Peach", "name": "topPhone"};
for (let p in product) {
     console.log(p +" : "+product[p]);  // type : phone
                                        // brand : Peach
                                        // name : topPhone
}
```

**La structure for (... of ...)**
La structure **for ... of ...** (introduite via la norme ECMAScript 6) permet d'accéder directement aux éléments de la liste.

```
let brands = ["Konia", "Peach", "Threestars"];
for (let e of brands) {
    console.log(e);  // Konia
                     // Peach
                     // Threestars
}
```

# 4 La programmation fonctionnelle en JavaScript

JavaScript se situe dans le paradigme de la programmation fonctionnelle : dans ce paradigme **les fonctions sont des éléments de premier plan** du langage.
En JavaScript, ce paradigme prend principalement deux incarnations fréquentes :
— une fonction peut-être passée en paramètre à une fonction hôte :
la fonction passée en paramètre est appelée **une fonction de callback** (ou fonction de rappel) ;
— une fonction (souvent appelée **une factory**) retourne une fonction.

## 4.1 Une fonction passée en paramètre (*fonction de callback*)

**L'utilisation des fonctions de callback (ou fonction de rappel) est un élément essentiel du langage JavaScript.**

Passer une fonction en paramètre d'une fonction hôte permet généralement de définir le code :
— qui doit être exécuté à la fin de l'exécution de la fonction (mais aussi en cours d'exécution de la fonction) ;
— qui consomme les données produites par la fonction.

Utiliser des fonctions de callback est un schéma de programmation très fréquent en JavaScript lors de l'**accès asynchrone à des données** (la fonction de rappel ne sera appelée que lorsque les données seront disponibles).

Au travers des deux méthodes **forEach()** et **map()** appliquées à des listes, nous donnons ci-après deux exemples de l'emploi des méthodes de callback.

### 4.1.1 Exemple avec la méthode *forEach()*

Voici un exemple d'utilisation d'une fonction de callback avec la méthode *forEach()* qui permet d'itérer sur ses éléments d'une liste.

```
let brands = ["Konia", "Peach", "Threestars"];
brands.forEach(function(e) {
    console.log(e); // Konia
                    // Peach
                    // Threestars
});
```

Une nouvelle syntaxe (norme ECMA6) permet d'utiliser la "flèche grasse" (*fat arrow*) pour implémenter la fonction de callback :

```
let brands = ["Konia", "Peach", "Threestars"];
brands.forEach((e) => { console.log(e); });
```

Dans la section suivante (chapitre 5.5) consacrée à la programmation objets, nous verrons comment est implémentée une telle fonction.

### 4.1.2 Exemple avec la méthode *map()*

Voici un autre exemple d'utilisation d'une fonction de callback avec la méthode *map()* qui permet d'appliquer une fonction sur chaque élément d'une liste puis de retourner cette liste modifiée.
Imaginons que nous disposions d'une collection qui liste des produits (représentés par des objets) mis dans le panier avec leurs prix et quantités.
Nous voulons modifier cette collection pour rajouter à chaque produit une nouvelle propriété nommée *subTotal* qui ait en valeur le nombre d'occurrences du produit multiplié par son prix.

```
function subTotal(e) {
   e.subtotal = e.price * e.quantity;
   return e;
}

var cart = [{"name":"topPhone", "price":1000, "quantity":2},
            {"name":"bigPhone", "price":700, "quantity":1}];
cart.map(subTotal).forEach(function(e) {
     console.dir(e);  // Affiche chaque objet modifié de la liste
});
```

Cet exemple provoque l'affichage suivant dans la console du navigateur :

```
{ name: 'topPhone', price: 1000, quantity: 2, subtotal: 2000 }
{ name: 'bigPhone', price: 700, quantity: 1, subtotal: 700 }
```

Dans la section suivante consacrée à la programmation objets, nous verrons comment est implémentée la fonction *map()*.

## 4.2 Une fonction retourne une fonction (*factory*)

Une fonction qui retorune une fonction est appelée une **factory**.

Dans l'exemple suivant (fichier *factory.html*), la fonction *manufacture()* permet de créer ("d'usiner", dont le nom de *factory*) les fonctions *phoneManufacture()* et *tabletManufacture()* qui permettent d'insérer un téléphone ou une tablette dans la liste appropriée.

*Attention, pour comprendre cet exemple, il est sans doute possible que vous soyez obligé de lire la section suivante sur le programmation par objets en JavaScript.*

```
var ProductType = function (type) {
   this.type = type;
   this.liste = new Array();

   this.productDisplay = function () {
      console.log(type+" :");
      for (let product of this.liste) console.log(product);
   }
}

var manufacture = function (ProductType) {
   return function set(product) { ProductType.liste.push(product); };
};

var phones = new ProductType('phone');
phones.liste.push({"name":'topPhone', "brand":"Peach"});

var phoneManufacture = manufacture(phones);
phoneManufacture({"name":"bigPhone", "brand":"Threestars"});
phones.productDisplay();

var tablets = new ProductType('tablet');
var tabletManufacture = manufacture(ipods);
tabletManufacture({"name":'bigTablet', "brand":"Threestarts"});
tablets.productDisplay();
```

Cet exemple affichera le résultat suivant dans la console du navigateur :

```
phones :
Object { name: "topPhone", brand: "Peach" }
Object { name: "bigPhone", brand: "Threestars" }
tablets :
Object { name: "bigTablet", brand: "Threestarts" }
```

# 5 La programmation objets avec JavaScript

## 5.1 Principes de la programmation objets avec JavaScript

Si quasiment tout est objet en Javascript, ce langage **ne gérait pas de classes** (contrairement par exemple aux langages JAVA, C++ ou Python), avant sa normalisation par la norme **ECMAScript 6** (en 2015, soit vingt ans après sa création).

La programmation objets avec JavaScript sera donc présentée en trois temps :
— la gestion classique des objets (sans utilisation de classes pour les instancier) ;
— l'introduction des classes (et d'un héritage simplifié) avec la norme *ECMAScript 6* (voir la section correspondante) ;
— l'utilisation d'extension du langage JavaScript pour gérer des classes, notamment avec *TypeScript* (voir le chapitre correspondant).

En restant sur leur gestion traditionnelle (sans classes), les objets peuvent être créés de deux façons :
— soit de manière littérale en utilisant les accolades ;
— soit en utilisant une **fonction constructrice** et en appelant cette fonction via l'instruction **new**.

L'héritage étant assuré par des objets particuliers appelés **prototypes** : ce type d'héritage est appelé **héritage par délégation**.

## 5.2 Les objets littéraux

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

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/`. Ces extensions seront présentées ultérieurement.

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

```
let topPhone = { "type": "phone", "brand": "Peach", "name": "topPhone"};
```

*En utilisant les accolades, l'objet que nous créons **hérite** du "méta objet" **Object**. Ce point sera détaillé ultérieurement.*

Il est important de remarquer qu'une propriété d'un objet peut avoir n'importe quelle valeur :
— une valeur booléenne ;
— une valeur scalaire (un entier, un réel, une chaîne de caractères...) ;
— une liste (tableau dynamique), par exemple `["Konia", "Peach", "Threestars"]` ;
— un objet ;
— le code implémentant une fonction ;
— ...

Il est aussi intéressant de noter que :
— même si théoriquement il n'est pas nécessaire d'encadrer les noms des propriétés par des simples ou doubles quotes, il est souvent préférable de le faire, certains parseurs l'exigeant ;
— une liste d'objets est appelée une *collection*.

## 5.3 L'héritage par chaînage de prototypes

### 5.3.1 La propriété *__proto__* de l'objet héritant

Un objet peut hériter des propriétés d'un autre objet via la propriété `__proto__`.

Imaginons que nous avons créé un objet nommé *PeachPhone* contenant le propriété *brand* : nous pouvons créer un nouvel objet nommé *topPhone* contenant le propriété *name* et **héritant des propriétés** contenues dans l'objet *PeachPhone* grâce à la propriété *__proto__* :

```
let PeachPhone = { "brand" : "Peach" };

let topPhone = {
  name : "topPhone",
  __proto__ : PeachPhone
};

console.dir(topPhone);  // Affiche :
                        // nom: "topPhone"
                        // __proto__ : Object

console.log(topPhone.brand);  // Affiche : Peach
```

Il est important de remarquer que *__proto__* est un pointeur sur l'objet hérité (une référence), et donc que si les propriétés de l'objet hérité (ou les valeurs de celles-ci) sont modifiées après la création de l'objet héritant, celui-ci héritera automatiquement des nouvelles propriétés/valeurs :

```
let PeachPhone = { "brand" : "Peach" };

let topPhone = {
  name : "topPhone",
  __proto__ : telephonePeach
};

PeachPhone.brand = "Pear";

console.log(topPhone.brand);  // Affiche : Pear (et non Peach)
```

### 5.3.2 La propriété *prototype* de l'objet hérité

Bien qu'un objet puisse hériter globalement d'un autre objet, il est conventionnel de ne le faire hériter que de la *part héritable* de cet objet. Cette part héritable est l'objet défini par la propriété *prototype*.

Voici l'exemple précédent où l'objet *PeachPhone* délocalise dans la propriété *prototype* ce qui peut être hérité (ainsi la propriété *comment* ne doit pas être héritée).

```
let PeachPhone = {
  "comment" : "cette marque est vraiment mystérieuse",
  "prototype" : { "brand" : "Peach" }
};

let topPhone = {
  name : "topPhone",
  __proto__ : PeachPhone.prototype
};

console.dir(topPhone);  // Affiche :
                        // nom: "topPhone"
                        // __proto__ : Object

console.log(topPhone.brand);  // Affiche : Peach
```

Dans le paragraphe suivant, nous verrons que la création d'un objet par appel d'une fonction constructrice utilisera systématiquement la propriété *prototype* de l'objet hérité (qui prendra la forme d'une fonction).

## 5.4 La création d'un objet par appel d'une fonction constructrice

En JavaScript (et avant l'application de la norme ECMAScript 6 qui crée les classes), il est bien plus commun de créer des objets via des **fonctions constructrices** et en utilisant l'opérateur **new**, que de créer des objets de manière littérale. En effet la fonction constructrice va servir de modèle pour définir les propriétés créées dans l'objet, le mot-clef **this** désignant dans celle-ci l'objet en cours de création.

Pour illustrer nos propos, nous allons définir une fonction nommée *phone()* qui reçoit en paramètres deux chaînes de caractères correspondant au nom de modèle et à la marque d'un téléphone. Cette fonction va retourner un nouvel objet qui possèdera deux propriétés nommées *name* et *brand* valuées avec les valeurs passées en paramètres.
Par ailleurs, la fonction *phone()* étant elle-même un objet, nous allons affilier à son prototype une méthode nommée *whoAmI()* qui sera héritée par tous les objets qu'elle aura créés (ce point sera discuté plus en détail après la présentation de notre exemple).

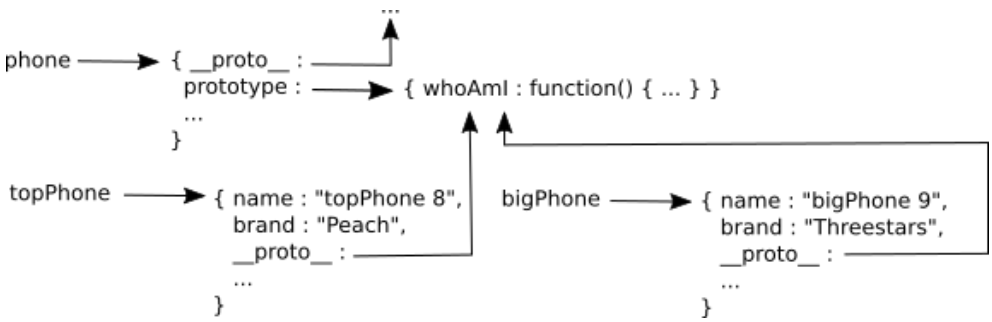
Dans l'exemple suivant, deux objets sont créés grâce à la fonction *phone* ; ces deux objets ont comme propriétés *name* et *brand* et utilisent la méthode *whoAmI()* associée au prototype de l'objet-fonction *phone* :

```
function phone (name, brand) {
   this.name = name;
   this.brand = brand;
}
phone.prototype.whoAmI = function() {
   console.log("Je suis le "+this.name+" de marque "+this.brand);
}

var topPhone = new phone("topPhone", "Peach");
var bigPhone = new phone("bigPhone", "Threestars");

topPhone.whoAmI();  // Je suis le topPhone de marque Peach
bigPhone.whoAmI();  // Je suis le bigPhone de marque Threestars
```

La fonction *phone*, comme toutes les fonctions en JavaScript, **est elle-même un objet** qui possède une propriété *prototype* référencée par les objets *topPhone* et *bigPhone* via leurs propriétés *__proto__* :




## 5.5 Deux exemples d'implémentations d'une méthode

Dans les exemples suivants, nous réimplémentons les méthodes **forEach()** et **map()** qui s'appliquent aux listes,

Voici un exemple de réimplémentation de la méthode *forEach()* qui accède à chaque élément d'une liste :

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

Voici un exemple de réimplémentation de la méthode *map()* qui applique une fonction à chaque élément d'une liste et renvoie la liste ainsi modifiée :

```
Array.prototype.map = function(treatment) {
   var results = [];
   for (let i in this) results.push(treatment(this[i]));
   return results;
}
```

## 5.6 La problématique de l'objet courant (*this*)

La manipulation de **this** qui désigne l'objet courant peut réserver quelques surprises, notamment quand le script JavaScript est exécuté au sein d'un navigateur. Il faut retenir trois principes fondamentaux qui permettront de les éviter :

- Initialement *this* désigne l'objet *Window* ;
- Dans une fonction constructrice invoquée par *new*, *this* désigne l'objet qui vient d'être créé par celle-ci.
  → cela est également vrai dans une méthode rattachée au *prototype* de cette fonction constructrice
- Dans une méthode d'un objet, *this* désigne celui-ci ;
- Mais dans une fonction de callback d'une méthode d'un objet, *this* désigne **Window**, à moins d'utiliser la syntaxe utilisant la *fat arrow* (voir ci-après).

Le code suivant chargé dans un navigateur positionne *this* à *Window* :

```
<html>
   <head>
      <script>
          console.log(this);  // Affiche : Window
      </script>
   </head>
</html>
```

Dans une fonction constructrice ou dans une méthode de son prototype, *this* désigne bien l'objet courant :

```
function fonction(name) {
     this.name = name;
     console.log(this);
}
fonction.prototype.methode = function() {
     console.log(this);
}

var objet = new fonction('objet1');   // Affiche : Object { name: "objet1" }
objet.methode();   // Affiche : Object { name: "objet1" }
```

Mais dans une fonction de callback, *this* ne désigne pas l'objet sur lequel est appliquée la méthode qui appelle cette fonction de callback :

```
function fonction(name) { this.name = name; }

fonction.prototype.methode = function(callback) {
    console.log(this);  // Affiche : Object { name: "objet1" }
    callback();
};

var objet = new fonction('objet1');
objet.methode(function() {
    console.log(this);  // Affiche : Window
});
```

Pour pallier ce problème, nous allons utiliser une des trois fonctions *call()*,*apply()* et *bind()*.

Les méthodes **call()**, **apply()** et **bind()** permettent d'appeler une fonction en associant son contexte à un objet particulier passé en premier paramètre de ces trois méthodes.
Les deux premières méthodes méthodes ne diffèrent que par la syntaxe de passage des paramètres, *bind()* crée une nouvelle fonction :

- avec *call()*, les paramètres de la fonction sont listés comme paramètres de *call* :
  `fonction.call(objet [, parametre1, parametre2, ...])`
- avec *apply()*, les paramètres de la fonction sont passés dans un tableau :
  `fonction.apply(objet [, [parametre1, parametre2, ...]]);`

**bind()** en revanche permet de créer une nouvelle fonction :
`nouvelleFontion = fonction.bind(objet [, parametre1, parametre2, ...])`

Reprenons notre exemple précédent : nous voudrions que la fonction de callback connaisse l'objet courant de la fonction qui appelle cette fonction de callback. Nous passons également un paramètre à la fonction de callback.

La première possibilité est d'utiliser la méthode **call()** :

```
function fonction(name) { this.name = name; }

fonction.prototype.methode = function(callback) {
    console.log(this);  // Affiche : Object { name: "objet1" }
    callback.call(this, "un paramètre");
};

var objet = new fonction('objet1');
objet.methode(function(param) {
  console.log("callback avec "+param+" et "+this);
     // Affiche : callback avec un paramètre et Object { name: "objet1" }
});
```

La deuxième possibilité est d'utiliser la méthode **apply()** :

```
function fonction(name) { this.name = name; }

fonction.prototype.methode = function(callback) {
    console.log(this);  // Affiche : Object { name: "objet1" }
    callback.apply(this, ["un paramètre"]);
};

var objet = new fonction('objet1');
objet.methode(function(param) {
  console.log("callback avec "+param+" et "+this);
     // Affiche : callback avec un paramètre et Object { name: "objet1" }
});
```

La troisième possibilité est d'utiliser la méthode **bind()** :

```
function fonction(name) { this.name = name; }

fonction.prototype.methode = function(callback) {
  console.log(this);  // Affiche : Object { name: "objet1" }
    let newFunction = callback.bind(this, "un paramètre");
    newFunction();
};

var objet = new fonction('objet1');
objet.methode(function(param) {
  console.log("callback avec "+param+" et "+this);
     // Affiche : callback avec un paramètre et Object { name: "objet1" }
});
```

## 5.7 L'héritage (là où les choses se compliquent)

En JavaScript "pur", l'héritage est implémenté par le chaînage de prototype.

Voici un exemple où la fonction constructrice *PeachPhone()* crée des objets correspondants à des téléphones de marque *Peach*. Cette fonction hérite du prototype de la fonction *phone()*.
Vous remarquerez l'utilisation de la méthode *call()* :

```
function phone(name) { this.name = name; }

phone.prototype.whoAmI = function() {
   console.log("Je suis le "+this.name+" de marque "+this.brand);
}

function PeachPhone (name) {
   this.brand = "Peach";
   phone.call(this, name);
}
PeachPhone.prototype = phone.prototype;

var topPhone = new PeachPhone("topPhone");
topPhone.whoAmI();  // Affiche : Je suis le topPhone de marque Peach
```

## 5.8 Le chaînage de méthodes

Comme dans tous les langages objets, il est commun en JavaScript d'enchaîner les méthodes comme suit : `object.method1().method2(). ... .methodn();`

Pour implémenter cela, il suffit que les méthodes renvoient l'objet sur lequel elles s'appliquent.
En voici un exemple :

```
function append(string) {  this.value += string; return this; }

function create(string) { this.value = string; }
create.prototype.append = append;

var initialValue = new create("2");
console.log(initialValue.append('Fast').append('4U').value); // 2fast4U
```

# 6 Les principaux apports de la norme ECMA6

## 6.1 La norme ECMAScript

**ECMAScript** est un ensemble de normes de programmation recommandées pour les langages de script. Ces normes sont principalement appliquées par les interpréteurs JavaScript (ceux des navigateurs et par Node.js).

Ces recommandations étant très nombreuses, nous ne détaillerons que quelques unes de celles définies en 2015 (norme ECMAScript 6), qui ont considérablement fait évoluer le langage, et sont largement diffusées depuis :

- le mot réservé **let**
- **l'interpolation** des variables dans les chaînes de caractères
- les **paramètres par défaut**
- la manipulation plus confortable des listes avec :
  - la structure itérative **for (... of ...)**
  - la méthode **includes()** pour tester la présence d'un élément
- l'utilisation des **fat arrow** pour simplifier l'écriture des fonctions de callback
- les **classes**

## 6.2 Le mot réservé *let*

Le mot réservé **let** permet de définir la portée de la variable qu'il introduit au bloc d'instructions courant.
Il est par exemple systématiquement utilisé pour les variables locales aux structures itératives :

```
for (let <variable> in ...) { ... }
for (let <variable> of ...) { ... }
```

Il est possible que soit soyez tenu à utiliser la directive `"use strict"` pour utiliser ce mot réservé.

## 6.3 L'interpolation de variables dans les chaînes

Pour manipuler des chaînes de caractères s'étendant sur plusieurs lignes et/ou interpolant des variables, il est désormais possible d'utiliser les **backquotes** et ainsi de créer des **template strings**.

En voici un exemple :

```
let v1 = 1; let v2 = 2; let v3 = 3;
let v4 = "partez";
console.log(`${v1} ${v2} ${v3} ... ${v4} ! `);
```

## 6.4 Les paramètres par défaut

Lors de la définition d'une fonction, des valeurs par défaut de paramètres optionnels peuvent être définies :

```
function test(a, b=0) {
   console.log(a+" et "+b);
}
test(1);  // affiche : 1 et 0
```

## 6.5 Une manipulation plus confortable des listes

### 6.5.1 La structure *for (... of ...)*

La structure itérative **for (let <variable> of <liste>)** permet d'accéder directement aux éléments d'une liste alors que la structure **for (let <variable> in <liste>)** n'instancie la variable locale à la boucle qu'avec les indices des éléments de la liste.

```
var brands = ["Peach", "Threestars", "Sony Ericsson"];
for (let e of brands) {
    console.log(e);  // Peach
                     // Threestars
                     // Sony Ericsson
}
```

### 6.5.2 La méthode *includes()*

La méthode **includes()** permet de tester si une liste contient un élément particulier (la méthode *indexOf()* ne permettant que de connaître l'indice de l'élément) :

```
if (brands.includes("Peach")) {
   console.log("Nous vendons aussi des marques fruits...");
}
```

## 6.6 L'opérateur "fat arrow" (=>)

La fat arrow (=>) est non seulement un raccourci syntaxique pour les fonctions de callback, mais permet aussi de conserver le contexte de l'objet courant (*this*) dans la fonction de callback.

Par exemple, l'écriture syntaxique de la fonction de callback passée en paramètre de la méthode *forEach()* peut être simplifiée.
Ainsi :

```
var liste = [1, 2, 3];
liste.forEach(function callback(valeur) {
   console.log(valeur);
});
```

devient :

```
liste.forEach((valeur) => { console.log(valeur) });
```

voire dans ce cas-ci (un seul paramètre) :

```
liste.forEach(valeur => { console.log(valeur) });
```

Outre cette simplification d'écriture, un point important est que le contexte (*this*) est conservé dans une fonction de callback utilisée dans une méthode de l'objet (attnetion, l'implémentation de cette fonctionnalité dépend de la version de votre interpréteur JavaScript).

### 6.7 Les classes

ECMAScript 6 définit les classes (via le mot clef **class**) et un héritage simple.
La classe possède un constructeur défini par la fonction *constructor()*, et invoqué par l'instruction **new**.

```
class Phone {
    constructor(name, brand) {
        this.name = name;
        this.brand = brand;
    }

    whoAmI() {
        console.log("Je suis le "+this.name+" de marque "+this.brand);
    }
}

let topPhone = new Phone("topPhone", "Peach");
topPhone.whoAmI();  // Je suis le topPhone de marque Peach
```

# 7 La programmation réactive, les observables et les promises

Les **observables** sont les éléments-clefs de la **programmation réactive**.

Un observable est n'importe quelle entité d'un programme qui peut produire un flux d'événements capturables au cours du temps : cela peut être un contrôle de l'interface qui est activé (par exemple un bouton qui est cliqué) ou une variable qui va changer de valeur (dans ce cas les événements sont associés à des valeurs signifiantes). Le but de la **programmation réactive** est de pouvoir associer **de manière asynchrone** un traitement aux valeurs émises par l'observable (le traitement est associé à l'observable via un **abonnement**). Plus particulièrement une fonction pourra être exécutée pour chaque valeur reçue, une autre en cas d'erreur, et enfin une dernière en cas de fin explicite du flux (les deux dernières fonctions pouvant être occultées). Ces fonctions sont les observateurs (*observers*) bien connus du patron de conception "observateur".

Ce paradigme est mis en œuvre en JavaScript via la bibliothèque **RxJS** (et à l'heure à laquelle j'écris cet ouvrage, en cours de normalisation sous ECMAScript).

La bibliothèque RxJS peut être utilisée comme module de Node.js ou comme un fichier à lier à un code HTML.
Pour l'utiliser comme module de *Node.js* (qui est présenté dans le chapitre "La plateforme Node.js"), il faut l'installer avec son gestionnaire de module *npm* avec la commande `npm install rxjs`.
Pour l'utiliser directement dans un code HTML, il faut également utiliser *npm* avec la commande `npm install -g rx-lite`, puis recopier le fichier *rx.lite.min.js* dans le répertoire où est localisé le code HTML.

Voici la création d'un observable dans un code exécuté avec Node.js :

```
var Rx = require('rxjs');
var source = Rx.Observable;
```

les méthodes suivantes (parmi beaucoup d'autres) pourront être utilisées sur un observable :
— **subscribe()** : abonne un traitement à l'observable
exemple : `source.subscribe(value => console.log(value))` → affiche chaque valeur reçue par l'observable
— **unsubscribe()** : désabonne le traitement attaché à l'observable
exemple : `source.unsubscribe()`
— *range()* : alimente l'observable avec une séquence d'entiers compris entre deux valeurs
exemple : `source.range(1, 10)` → alimente l'observable avec les entiers de 1 à 10
— *interval()* : alimente l'observable avec un signal délivré à intervalle de temps donné
exemple : `source.interval(1000)` → l'observable reçoit un signal toutes les secondes
— *take()* : permet de ne retenir qu'un ensemble de valeurs parmi celles générées :
exemple : `source.interval(1000).take(10)` → l'observable reçoit un signal toutes les secondes pendant 10 secondes

Dans les trois exemples suivants, nous allons programmer un :
— observable sur un bouton ;
— observable sur le changement de valeur d'un entier ;
— observable sur un intervalle de temps.

Le premier exemple est un code HTML à charger dans votre navigateur. Les deux exemples suivants sont à exécuter via *Node.js* par la commande `node <nom du programme>`.

Bien entendu, l'utilisation d'observables sera omniprésent sous Angular.

## 7.1 Premier exemple : un observable sur un bouton

Voici un premier exemple qui met en œuvre la bibliothèque *RxJS* en la liant directement à un code HTML. Dans cet exemple nous utilisons aussi la bibliothèque *JQuery* pour accéder à un élément du DOM (en l'occurrence un bouton).

```
<html>
  <head>
    <meta charset="utf-8" />
    <script src="rx.lite.min.js"></script>
    <script src="jquery.min.js"></script>
    <script>
      $().ready(function(){
         var button = $('#testButton');
         Rx.Observable.fromEvent(button, 'click')
                      .subscribe(e => console.log("click"));
      });
    </script>
  </head>
  <body>
    <button id="testButton"> Test button </button>
  </body>
</html>
```

A chaque sélection du bouton, le message s'affiche sur la console.

## 7.2 Deuxième exemple : un observable sur un entier

Pour ce second exemple, nous créons un observable sur un entier dont la valeur va varier de 1 à 5.
Nous abonnons à cet observable trois observateurs (*observers*) :

— un observateur qui affiche chaque valeur reçue ;
— un observateur qui affiche un message d'erreur si celui-ci est émis ;
— un observateur qui affiche un message à la fin du flux.

```
var Rx = require('rxjs');
var source = Rx.Observable.range(1, 5);

var subscription = source.subscribe(
      value => console.log(value),
      err => console.log("Error : "+e),
      () => console.log("And that's all folks !"));
```

Ce code affichera :

```
1
2
3
4
5
And that's all folks !
```

Pour rappel, la syntaxe précédente utilise les *fat arrows*. En voici une version plus conservatrice qui n'utilise pas cet opérateur :

```
var subscription = source.subscribe(
       function(value) { console.log(value); },
       function(err) { console.log("Error "+e); },
       function() { console.log("And that'all folk !"); });
```

## 7.3 Troisième exemple : un observable sur un timer

Pour ce troisième exemple, nous créons un observable qui va compter les dix premières secondes en affichant "Top" sur la console :

```
var Rx = require('rxjs');

function traitement() { console.log("Top"); }

const sub = Rx.Observable.interval(1000)
              .subscribe(e => traitement());
setTimeout(() => sub.unsubscribe(), 10000);
```

Il est à remarquer que le premier "Top" s'affichera à plus d'une seconde après le début de l'exécution du script du temps de l'initialisation de l'observable.

## 7.4 Les *Promises*

Les promises (des promesses...) peuvent dans certains cas remplacer les observables, bien qu'elles soient en fait plus utilisées pour la meilleure lisibilité syntaxique qu'elles apportent, que pour la gestion asynchrone de données. En ce qui concerne celle-ci, la différence majeure entre les observables et les promises est que les promises n'attendent qu'un seul événement (et non un flux d'événements) ce qui est assez limité.
Il faut également noter les différences suivantes :

- en cas d'échec, l'abonnement à un observable est renouvelable ;
- et les observables sont également annulables (le désabonnement est possible via la méthode *unsubscribe()* ; les promises dans leur implémentation ES6 ne le sont pas (mais peuvent l'être dans d'autres implémentations).

Voici un exemple de création d'un objet *Promise*.

```
let p = new Promise(function(resolve, reject){ resolve(1); })
p.then(function(valeur){ console.log("étape 1 : "+valeur); return valeur+1; } )
 .then(function(valeur){ console.log("étape 2 : "+valeur); });
```

Ce code affichera `Etape 1 : 1`, puis `Etape 2 : 2`.

Voici le même code utilisant des fat arrows :

```
let p = new Promise((resolve, reject) => resolve(1) )
p.then(valeur => { console.log("étape 1 : "+valeur); return valeur+1; } )
 .then(valeur => console.log("étape 2 : "+valeur) );
```

Pour revenir à la meilleure lisibilité syntaxique apportée par les promises, imaginons que nous voulions à partir du nom d'un produit connaître sa marque, puis à partir d'une marque connaître le dernier produit acheté de cette marque-là. Nous supposons que les fonctions *getBrand()* et *getLastPurchaseByBrand()* remplissent respectivement ces rôles.

La syntaxe classique utilisant les callbacks serait celle-ci :

```
getBrand(productName, function(brand) {
   getLastPurchaseByBrand(brand, function(purchase) {
      ...
   });
});
```

celle utilisant les *fat arrows* celle-là :

```
getBrand(productName, brand => { getLastPurchaseByBrand(brand, purchase => { ... }) });
```

et enfin voici celle qui serait mise en œuvre par une promise :

```
getBrand(productName)
  .then(function(brand) {
     return getLastPurchaseByBrand(brand);
  })
  .then(function(purchase) {
     ...
});
```

# 8 La gestion d'événements :

Le plus ancien usage de Javascript est de gérer au niveau du client (généralement le navigateur) des événements utilisateurs (dans le but par exemple d'intercepter la soumission de formulaires pour vérifier que tous les champs sont renseignés, l'affichage d'informations supplémentaires dans la page web lors d'un clic souris...).

La gestion des événements passe par l'association d'écouteurs d'événements à des balises HTML, dont voici quelques exemples (il y en a beaucoup plus) :

- `onClick` : écouteur sur l'événement "clic souris" :
  exemple sur un clic souris sur un item d'une liste non déroulante : `<li onClick="javascript:fonction(...)">`
- `onChange` : sélection d'un item d'une liste déroulante ;
- `onLoad` : lors du chargement de la page (accompagnant la balise `<body>`) pour exécuter du code JavaScript après que tous les éléments du DOM aient été créés ;
- `onUnLoad` : lors du déchargement de la page (idem) ;
- ...

A ces événements sont associées des fonctions javascript généralement codées dans un bloc `<script></script>` situé dans la partie d'en-tête de la page web ou externalisées dans des fichiers d'extension `.js`. Les fonctions Javascript ainsi invoquées vont pouvoir interagir avec les informations stockées dans le **DOM (Document Object Model), (voir chapitre suivant)** qui instancie en mémoire du navigateur tous les éléments de la page web (écrite dans un langage à balises), soit principalement :

- **le noeud de type document** qui correspond à la racine du DOM ;
- **les noeuds de type élément** qui correspondent aux balises du document ;
- **les noeuds de type attribut** qui correspondent aux attributs qui accompagnent les balises ;
- **les noeuds de type texte** qui correspondent aux chaînes de caractères.

Voici l'exemple du changement de la taille d'une image lors d'un clic et via la référence à l'objet sélectionné (**this**) :

```
<html>
  <head>
    <script>
      function changeTailleImage(image) {
         image.style.width  = "100px" ;
         image.style.height = "100px" ;
      }
    </script>
  </head>
  <body>
     <img src="IMAGES/cliches_d_amour.jpg" width="50px" height="50px"
          onClick="changeTailleImage(this)" />
  </body>
</html>
```

Une autre possibilité est de mettre en œuvre une méthode de l'objet *document* qui va sélectionner le ou les éléments du DOM qui doivent être accédés ou modifiés :

- document.**getElementById()** : sélection d'un élément par son identifiant (attribut **id**) ;
- document.**getElementsByName()** : sélection d'éléments par leur nom (attribut **name**) ;
- document.**getElementsByTagName()** : sélection d'éléments par le nom de la balise.

Voici l'exemple du changement de la taille de toutes les images au bout de 5 secondes :

```
<html>
  <head>
    <script>
      function changeTailleImages() {
        var images = document.getElementsByTagName("img");
        console.log("Nombre d'images : "+images.length);
        for (var i = 0; i < images.length; i++) {
             images[i].style.width  = "100px" ;
             images[i].style.height = "100px" ;
        }
      }
    </script>
  </head>
```

```
  <body onLoad="setTimeout('changeTailleImages()', 5000)">
     <img src="IMAGES/cliches_d_amour.jpg" width="50px" height="50px" />
     <img src="IMAGES/forever_and_ever.jpg" width="50px" height="50px" />
  </body>
</html>
```

Il est aussi possible d'associer un écouteur d'événements à un élément du DOM déjà créé via la méthode *addEventListener()* :

```
<html>
  <head>
    <script>
      function changeTailleImage() {
         this.style.width  = "100px" ;
         this.style.height = "100px" ;
      }

      function associationEcouteur() {
         document.getElementById("image1").addEventListener("click", changeTailleImage);
      }
    </script>
  </head>
  <body onLoad="associationEcouteur()">
     <img id="image1" src="Cliches_d_amour.jpg" width="50px" height="50px" />
  </body>
</html>
```

# 9 Manipulation du DOM :

**D'un point de vue opératoire** (et non pas d'un point de vue modèle ;)) le **DOM (Document Object Model)** est la hiérarchie d'objets correspondant aux entités (balises, attributs...) de la page web et instanciés à partir du modèle de classes que votre navigateur possède. Modifier le DOM impacte donc automatiquement la représentation de la page web.

Pour modifier le DOM, nous pouvons utiliser soit l'**API DOM de JavaScript** qui est la bibliothèque "de base" ou la bibliothèque **JQuery** de plus haut niveau.

## 9.1 Avec l'API DOM de Javascript :

Voici quelques fonctions de l'API DOM de Javascript :

- Informations sur les nœuds de l'arbre DOM :
  - `noeud.childNodes` : collections des fils du nœud (attribut `length` pour connaître le nombre) ;
  - `noeud.childNodes.length` : nombre de nœuds fils
  - `noeud.nodeName` : nom du nœud de type élément (balise) ou de type attribut ;
  - `noeud.nodeType` : type du nœud (ne pas se servir de cela pour tester les attributs !) :
    - 1 : nœud élément (correspondant à une balise) ;
    - 3 : nœud texte ;
    - 9 : nœud document.
  - `noeud.nodeValue` : valeur du nœud

- Informations sur les nœuds de type attribut :
  - `noeud.hasAttributes()` : teste si un nœud de type élément possède des attributs ;
  - `noeud.attributes` : collection des attributs portés par le nœud ;

- Création d'un nouveau nœud dans le DOM :
  - `document.createElement(...)` : création d'un nœud élément (correspondant à une balise) ;
  - `document.createTextNode(...)` : création d'un nœud texte ;
  - `noeud_texte.appendData(...)` : ajout d'une chaîne de caractères au contenu d'un nœud texte ;
  - `noeud.appendChild(...)` : affiliation d'un nœud à un autre nœud père de l'arbre DOM ;
  - `noeud.setAttribute(nom, valeur)` : création/modification de la valeur d'un attribut

## 9.2 Avec la bibliothèque JQuery :

La bibliothèque JQuery se lie à votre code (ici après avoir été téléchargée) → `https://jquery.com/`
via la balise `<script>` : `<script src="jquery.min.js" type="text/javascript" />`

Il est aussi à remarquer que JQuery peut être installée via **npm** le gestionnaire de module de la plateforme JavaScript **Node.js**.

Des nœuds du DOM peuvent être sélectionnés en utilisant une syntaxe `$(sélecteur)` qui reprend celle de CSS.
Cette sélection renvoie une liste de nœuds (un *node set*). Voici les sélecteurs les plus courants :

```
élément (balise)          $("td")
#id                       $("#maDiv")
.class                    $(".highlight")
parent>enfant             $("td>table") : sélection de tous les éléments table fils d'un élément td
*                         $("td>*")
:first                    $("a:first")
:last                     $("div:last")
:eq()                     $("div:eq(1)") : sélection de la deuxième division
:contains                 $("a:contains('important')")
élément[attribut=valeur]  $("[name='info']") : sélection de tous les éléments dont le nom est égal à 'info'
                          $("input[type=text]")
élément[attribut!=valeur] $("[id!='param']") : sélection de tous les éléments dont l'id est différent de 'param'
élément[attribut^=valeur] $("[id^='param']") : sélection de tous les éléments dont l'id commence par 'param'
élément[attribut*=valeur] $("[id*='param']") : sélection de tous les éléments dont l'id comprend 'param'
élément[attribut*=valeur] $("[id$='param']") : sélection de tous les éléments dont l'id finit par 'param'
```

Par ailleurs :
`$()` sélectionne le **nœud document** de l'arbre DOM.
`$(this)` sélectionne le **nœud courant**

et des tests peuvent être appliqués sur les éléments du *node set* avec la méthode **is()** : `if ($("#zoom").is(":checked"))`

**Exemple :**

Dans cet exemple :
- `ready()` n'exécutera le code JQuery qu'à partir du moment où le DOM correspondant à la page HTML aura été complètement instancié.
- toutes les balises filles de `<ul>` (a priori des `<li>`) qui contiennent la chaîne de caractères "item" sont sélectionnées et le nom de la balise ainsi que la valeur de son identifiant sont affichés
  (vous remarquerez que l'objet courant peut être sélectionné soit par `this` ou par `$(this)`)
- toutes les ancres sont soulignées si elles contiennent la chaîne de caractères "important".

```
$().ready(function() {
   $("ul>*:contains('item')").each(function() {
       console.log(this.tagName+" "+$(this).attr("id"));
   });
   $("a:contains('important')").css("text-decoration", "underline");
});
```

### 9.2.1 Insertion de nouveaux nœuds dans le DOM :

|

JQuery nous propose toute une série de méthodes pour insérer un nouveau nœud dans le DOM :
(Il y en a beaucoup plus que celles listées ci-après) :
- `append()` : insertion d'un élément à la fin de la cible
- `appendTo()` : idem (voir exemple)
- `prepend()` : insertion d'un élément au début de la cible
- `prependTo()` : idem (voir exemple)

Différents emplois des fonctions d'insertions :

```
$('sélecteur').append('nouveau texte');
$('nouveau texte').appendTo('sélecteur');
$('sélecteur').prepend('nouveau texte');
$('nouveau texte').prependTo('sélecteur');
```

Dans l'exemple suivant, une collection d'objets appelée *data* est utilisée pour construire une liste HTML qui est ensuite insérée après le dernier élément de la balise `<body>` (c'est la propriété *nom* des objets de la collection *data* qui donnera la valeur des items de la liste) :

```
var liste = "<ul>";
$.each(data, function(indiceObjet, objet) {
       liste += "<li>"+objet['nom']+"</li>";
});
liste += "</ul>";
$("body").append(liste);
```

### 9.2.2 Manipulation des attributs et des valeurs d'un nœud du DOM :

Voici les méthodes les plus fréquentes proposées par JQuery :
- accès à la valeur d'un attribut : `$('sélecteur').attr(nomAtribut);`
- association d'un attribut à un élément : `$('sélecteur').attr(nomAtribut, valeurAttribut);`
- accès à la valeur d'un élément : `$('sélecteur').val();`

### 9.2.3 Ecouteurs d'événements :

JQuery propose un certains nombre de méthodes comme `click()` ou `live()` (à découvrir...) pour associer des écouteurs d'événements à des éléments de la page web.
Voici l'exemple de la manipulation d'un attribut d'un élément qui vient d'être cliqué :

```
$('sélecteur').click(function(){
    $(this).attr(nomAtribut, valeurAttribut);
});
```

# 10 AJAX : Asynchronous Javascript and XML

Javacript nous offre la possibilité de solliciter un serveur pour que celui-ci renvoie des données (généralement formatées en XMl iou plus encore en JSON) de manière asynchrone. Ces données permettent dans un usage fréquent de modifier dynamiquement le DOM et donc l'affichage de la page web sans que celle-ci soit complètement rechargée à partir du serveur.

## 10.1 En javascript "pur" :

Les objets instanciés à partir de la fonction "constructrice" `XMLHttpRequest` permettent d'ouvrir une connexion asynchrone avec le serveur. Une fonction de callback est appelée lors des différentes étapes du chargement des données (principalement formatées en `XML` ou en `JSON` format de sérialisation d'objets JavaScript).

Le code suivant donne un exemple sur l'importation de données formatées en XML contenues dans le fichier *catalogue_musiques.xml*.

- `setTimeout('chargementXML()', 300);` permet d'attendre que tous les éléments de l'arbre DOM soient instanciés c'est une méthode maladroite qui sera beaucoup mieux gérée en `JQuery`.
- `connexion.readyState` indique le statut courant de la connexion ;
- `contenu` est une référence sur l'élement `document` de l'arbre DOM ;
- `var contenu=connexion.responseText` aurait permis de récupérer une chaîne de caractères.

```
var connexion;

function chargement() {
    if (connexion.readyState == 4) {
        // console.log("Le chargement des données est terminé !");
        var contenu=connexion.responseXML;
        var balise_top_level = contenu.childNodes.item(0);
        for (var num=0; num < balise_top_level.childNodes.length; num++) {
             console.log(balise_top_level.childNodes.item(num).toString());
        }
    }
}

function chargementXML() {
   if (window.XMLHttpRequest) {
       connexion = new XMLHttpRequest();
       if (connexion != 0) {
           connexion.onload = null;
           connexion.open("GET", "catalogue_musiques.xml", true);
           connexion.onreadystatechange = chargement;
           connexion.send(null);
       }
   }
   else { alert('La connexion n a pu être initiée !'); }
}

setTimeout('chargementXML()', 300);
```

**L'importation de données via AJAX ne se programme quasiment plus par du Javascript pur : l'utilisation de bibliothèque telles que `JQuery` est plus confortable notamment pour gérer les problèmes de cross-browsing.**

## 10.2 Avec la bibliothèque JQuery :

La bibliothèque JQuery se lie à votre code (ici après avoir été téléchargée) via la balise `<script>` :

```
<script src="jquery.min.js" type="text/javascript" />
```

JQuery propose deux fonctions pour opérer des téléchargements de données et une plus générique pour downloader ou uploader des données :

— `$.get()` : pour télécharger des données notamment en XML ;
— `$.getJSON()` : pour télécharger spécifiquement des données sérialisées en JSON.
— `$.ajax()` : la fonction la plus générique.

Attention : ces transferts de données sont **asynchrones**.

### 10.2.1 Téléchargement de données XML :

Dans ce schéma de programmation, la méthode *find()* permet de filtrer une balise particulière :

```
$.get(<fichierXML>, function(data) {
     data.find(...).each(function() {
          ...
     });
});
```

### 10.2.2 Téléchargement de données JSON :

Dans ce schéma de programmation utilisant la fonction *$.getJSON()* :

— `entryIndex` correspond au numéro d'ordre de chaque objet de la collection ;
— `entry` correspond à chaque objet de la collection.

```
$.getJSON(<fichierJSON>, function(data) {
     $.each(data, function(entryIndex, entry) {
          ...
     });
});
```

### 10.2.3 La fonction générique :

La fonction **$.ajax()** permet de spécifier la méthode HTTP, et donc en utilisant POST (ou PUT), de transmettre des données dans le corps du message envoyé au serveur.

Voici un exemple d'envoi de données au serveur en POST :

```
$.ajax({
  method: "POST",
  url: <URL>,
  data: { <propriétés de l'objet à transmettre> }
})
.done(function( msg ) {
          alert( msg );
});
```

# 11 Interfaçage de Maps :

## 11.1 Avec Google Maps :

**L'API Google Maps est devenue payante et ne sera pas utilisée en TP.**

La bibliothèque GoogleMap se lie à votre code via la balise `<script>`.

Une clef d'identification doit être demandée à l'URL suivante :

```
https://developers.google.com/maps/documentation/javascript/get-api-key?hl=Fr
```

Une fois la clef obtenue, voici le lien à effectuer :

```
<script src="https://maps.googleapis.com/maps/api/js?key=<YourKey>&callback=initMap"
        async defer></script>
```

### 11.1.1 Création d'une division pour afficher la carte :

Avant toute chose, il faut créer une division d'accueil dans votre code HTML :

```
<div id="map_canvas" style="width:100%; height:100%"></div>
```

### 11.1.2 Création de la carte :

La création de la carte nécessite au minimum le paramétrage :
— des coordonnées sur lesquelles elle va être centrée ;
— de son type (ici une carte routière)
— et la division d'accueil dans la page HTML.

```
var latlng = new google.maps.LatLng(43.6111, 3.87667);
var myOptions = { zoom: 15,
                  center: latlng,
                  mapTypeId: google.maps.MapTypeId.ROADMAP
};
var map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
```

### 11.1.3 Création d'un marqueur :

La création d'un marqueur nécessite seulement le paramétrage des coordonnées où il va se situer et la map d'appartenance :

```
var latlng = new google.maps.LatLng(43.6111, 3.87667);
var marker = new google.maps.Marker({ position: latlng,
                                      map: map,
                                      title: "Montpellier",
                                      draggable: false
                                    });
```

Une popup peut être liée au marqueur :

```
var infowindow = new google.maps.InfoWindow({content: 'Un joli petit message...'});
google.maps.event.addListener(marker, 'click', function() {
                                infowindow.open(map,marker);
                              });
```

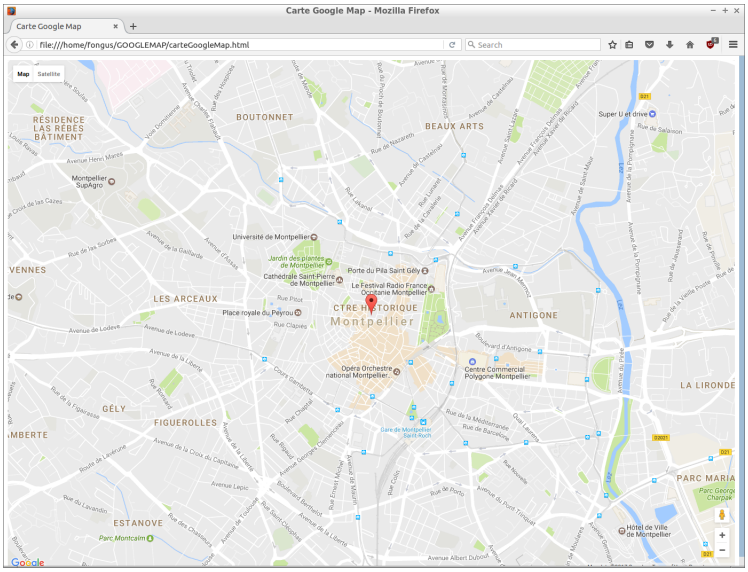
Voici donc le code créant une carte centrée sur la ville de Montpellier (n'oubliez pas d'insérer votre clef dans l'URL de l'API Google Map) :

```
<!DOCTYPE html>
<html>
   <head>
      <title> Carte Google Map </title>
      <meta charset="utf-8" />
      <style type="text/css">
        html { height: 100%; }
        body { height: 100%; }
      </style>
      <script src="https://maps.googleapis.com/maps/api/js?key=<YourKey>
                        &callback=initMap" async defer></script>

      <script type="text/javascript">
           function initMap() {
              var latlng = new google.maps.LatLng(43.6111, 3.87667);
              var myOptions = { zoom: 15,
                                center: latlng,
                                mapTypeId: google.maps.MapTypeId.ROADMAP
                              };
              var map = new google.maps.Map(document.getElementById("map_canvas"),
                                            myOptions);
              var marker = new google.maps.Marker({ position: latlng,
                                                    map: map,
                                                    title: "Montpellier",
                                                    draggable: false
                                                  });
           }
      </script>
   </head>
   <body>
         <div id="map_canvas" style="width:100%; height:100%"></div>
   </body>
</html>
```

Le code est assez limpide. Seules trois méthodes de l'API sont utilisées :
— la méthode *google.maps.Map()* crée une carte rattachée à une `<div>` HTML ;
— la méthode *google.maps.LatLng()* crée un objet spécifiant une coordonnée géographique (latitude et longitude) ;
— la méthode *google.maps.Marker()* crée un marqueur.

Vous remarquerez aussi que le paramètre *callback* de l'URL de la Google API référence la fonction qui initie la carte.



## 11.2 Avec OpenLayers et OpenStreetMap (OSM) :

**OpenLayers** est une bibliothèque qui permet (entre autres) d'exploiter la base de données géographique **OpenStreetMap**.

La **bibliothèque OpenLayers** (`http://openlayers.org/download/`) se lie à votre code via les balises suivantes (à recopier du site) :

```
<script src="https://cdn.rawgit.com/openlayers/openlayers.github.io/master/en/v6.0.1/build/ol.js"></script>
<link rel="stylesheet" href="https://cdn.rawgit.com/openlayers/openlayers.github.io/master/en/v6.0.1/css/ol.css">
```

### 11.2.1 Création d'une division pour afficher la carte :

Avant toute chose, il faut créer une division d'accueil dans votre code HTML (ici liée à une classe) :

```
<div id="map" class="map"></div>
```

associée au style suivant :

```
<style>
   .map {height: 100%; width: 100%; }
</style>
```

### 11.2.2 Création de la carte :

La création de la carte nécessite le même paramétrage que pour *Google Maps* (type de la carte, coordonnées et niveau de zoom). Ce paramétrage se fait via la création d'un objet à partir la fonction constructrice *ol.Map()* :

```
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([3.87667,43.6111]),
          zoom: 15 // max 11 pour vue satellitaire (sat)
      })
});
```

Vous remarquerez le lien sur le fournisseur de tuiles (tiles) cartographiques : OSM, càd OpenStreetMap.

### 11.2.3 Création d'un marqueur :

La création d'un marqueur nécessite **la superposition d'un layer qui va héberger l'image du marqueur**.
Sa gestion est plus difficile que via *Google Maps* car il faut gérer à la main la popup qui va y être associée (dans cet exemple la popup est une `<div>` affichant un message texte) :

L'image du marqueur :

```
<img id="marker" src="IMAGES/marker.png" widh="50" height="50" onClick="switchMarker();" />
```

La popup :

```
<div id="popup" style="display:none; color:red; width:100px; height:100px">
     Un joli petit message...
</div>
```

La gestion du marqueur :

```
var marker = document.getElementById('marker');
map.addOverlay(new ol.Overlay({
    position: ol.proj.fromLonLat([3.87667,43.6141]),
    element: marker
}));

var popup = document.getElementById('popup');
map.addOverlay(new ol.Overlay({
        offset : [0, -35],
        position: ol.proj.fromLonLat([3.87667,43.6141]),
        element: popup
}));
function switchMarker() { (popup.style.display == "none" ? popup.style.display = "block" :
                                                           popup.style.display = "none") };
```

*A vous de rassembler tous ces morceaux de codes...*

# 12 Création d'interfaces graphiques avec la bibliothèque D3.js :

Javascript peut manipuler la balise HTML5 `<canvas>` qui offre un support à du dessin vectoriel.
Cela-dit, je vous recommande d'utiliser la blibliothèque **D3.js** → `https://d3js.org/` pour créer un rendu graphique élaboré. La philosophie de D3 (*Data-Driven Documents*) est de générer des éléments graphiques en **SVG** à partir de collections d'objets.

La bibliothèque D3 se lie à votre code (ici après avoir été téléchargée) via la balise `<script>` :

```
<script src="d3.min.js" type="text/javascript" />
```

Le principe fondamental de D3 est de générer des éléments graphiques SVG, par exemple des cercles (`<circle r="..." x="..." y="..." />`) par rapport à une collection d'objets précisant les propriétés de ces éléments.

## 12.1 Aperçu du langage SVG

Le langage balisé **SVG** (*Scalable Vector Graphics*), normalisé par le W3C, permet de créer des graphiques 2D vectoriels. Il permet également de contrôler (via des événements), et d'animer les éléments graphiques ainsi créés. C'est un langage très complet, les interactions entre éléments graphiques étant scriptables en JavaScript.

Par exemple :

un rectangle peut être créé par la balise `rect` :

```
<rect x="<abscisse du coin supérieur gauche>"
      y="<ordonnée du coin supérieur gauche>"
      width="<largeur>" height="<hauteur>" stroke="<couleur de la bordure>"
      fill="<couleur de remplissage>" />
```

un cercle par le balise `circle` :

```
<circle cx="<abscisse du centre>" cy="<ordonnée du centre>" r="<rayon>"
        stroke="<couleur de la bordure>" fill="<couleur de remplissage>" />
```

un segment de droite par la balise `line` :

```
<line x1="<abscisse du point de départ>" y1="<ordonnée du point de départ>"
      x2="<abscisse du point d'arrivée>" y2="<ordonnée du point d'arrivée>"
      stroke="<couleur de la bordure>" fill="<couleur de remplissage>" />
```

une suite de segments de droites par la balise `polyline` :

```
<polyline points="<liste de coordonnées>" stroke="<couleur de la bordure>" fill="<couleur de remplissage>" />
```

un tracé arbitraire par la balise `path` :

```
<path d="<liste de coordonnées et éléments de contrôle de l'affichage>"
      stroke="<couleur de la bordure>" fill="<couleur de remplissage>" />
```

Voici un exemple de balise `<path>` :

```
<path d="M108,62 L90,10 70,45 50,10 32,62" style="stroke:black; fill:none" />
```

dans lequel :
— M signifie *MoveTo* ("saut" au point de coordonnées absolues qui suit) ;
— L signifie *LineTo* (les coordonnées suivantes sont reliées par des segments de droites).
D'autres éléments de contrôle comme *C* ou *A* permettent d'interpoler une courbe passant par les coordonnées qui sont listées. Par ailleurs, l'utilisation de ces caractères en minuscules impliquerait l'utilisation de coordonnées relatives.

La balise `g` permet de grouper des éléments graphiques pour créer un élément graphique composite qui pourra être réemployé avec la balise `g`.
Bien entendu, tous les éléments graphiques créés par les balises SVG peuvent être identifiés par un attribut *id*.

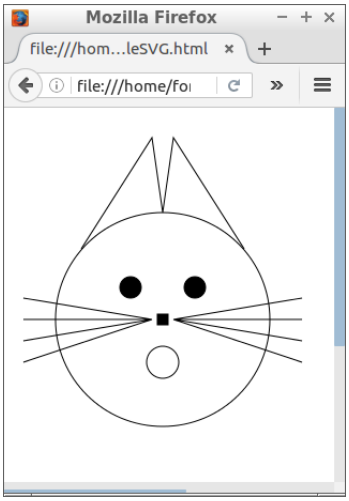
Des opérations de changements d'échelle, de translation et de rotation peuvent être appliquées grâce à l'attribut *transform* dont la valeur est régie par les fonctions suivantes :
— la fonction *scale()* permet un changement d'échelle
elle permet aussi d'effectuer des opérations de mirroring ;
— la fonction *translate()* permet d'effectuer des rotations ;
— la fonction *rotate()* permet d'effectuer des translations.

Voici un exemple classique de code SVG :

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
     width="500" height="500">

<circle cx="140" cy="190" r="100" style="stroke:black;fill:none"/>
<circle cx="110" cy="160" r="10" style="stroke:black;fill:black"/>
<circle cx="170" cy="160" r="10" style="stroke:black;fill:black"/>
<circle cx="140" cy="230" r="15" style="stroke:black;fill:none"/>
<rect x="135" y="185" width="10" height="10" style="stroke:black;fill:black"/>
<g id="vibrisses">
   <path d="M150,190 L270,170 M150,190 L270,190 M150,190 270,210 M150,190 270,230"
         style="stroke:black;"/>
</g>
<use xlink:href="#vibrisses" transform="scale(-1 1) translate(-280 0)" />
<polyline points="216 124, 150 20, 140 90, 130 20, 64 124"
          style="stroke:black; fill:none" />
</svg>
```

Dont voici le magnifique résultat :




## 12.2 Génération d'éléments graphiques associés aux objets d'une collection

Le principe fondamental de D3 est de générer des objets graphiques SVG (par exemple des cercles) par rapport à une collection d'objets JavaScript précisant tout ou partie de leurs propriétés (par exemple des objets JavaScript spécifiant les coordonnées de cercles).

Cette association se fait en quatre étapes :

— la sélection (ou la non sélection) d'éléments graphiques du DOM (déja créés) : cette étape est mise en œuvre par la méthode **selectAll(<sélecteur>)**
les sélecteurs ont une syntaxe proche de celle des CSS (voir le paragraphe suivant)
(si aucun sélecteur n'est passé en paramètre à cette méthode, aucun élément graphique n'est sélectionné) ;

— l'association logique entre les éléments sélectionnés du DOM et une collection d'objets JavaScript (un *dataset*) : cette association est effectuée rang par rang (premier élément du DOM avec le premier *datum* du dataset...)
cette étape est mise en œuvre par la méthode **data()**

— par rapport à l'association précédemment effectuée, la sélection des objets qui doivent être gérés ; cette étape est mise en œuvre par les méthodes :
  — **exit()** : renvoie les objets du dataset qui correspondent (rang à rang) aux éléments du DOM sélectionnés
  — **enter()** : renvoie les objets du dataset qui n'ont pas trouvé de place dans les éléments sélectionnés du DOM : ces éléments veulent entrer en scène...

— la stratégie d'association entre les éléments graphiques sélectionnés et la collection : cette étape est mise en œuvre par les méthodes :
  — enter().**append()** : pour les nouvelles données à prendre en compte, création de nouveaux éléments graphiques dans le DOM ;
  — exit().**remove()** : suppression des anciens éléments graphiques et construction de nouveaux éléments associés aux données du dataset de mêmes rangs.

## 12.3 Sélection et modification d'éléments du DOM

La sélection d'élément(s) du DOM peut être opérée par les méthodes *select()* et *selectAll()* qui s'appliquent sur l'objet *d3* de plus haut niveau (elle pourrait aussi se faire via JQuery).

La méthode *select()* permet de sélection un élément du DOM. En voici différents exemples :

```
let body = d3.select("body")
svg.select("#monId")  -> sélection de l'élément d'id "monId"
d3.select(this)  -> sélection de l'élément courant
```

La méthode *selectAll()* permet de sélectionner un ensemble d'éléments du DOM :

```
d3.selectAll("h3")  -> sélection des éléments correspondants à la balise "h3"
d3.selectAll(".maClasse")  -> sélection des éléments associés à la classe CSS "maClasse"
d3.selectAll()  -> renvoie une liste vide
```

### 12.3.1 Ajout d'éléments graphiques

Dans l'exemple suivant, aucun élément graphique préexistant n'est sélectionné : il y a autant de cercles qui sont créés que d'objets dans le dataset.
(Le rayon des cercles est fixé dans le code, mais leurs positionnements dépendent du dataset)

```
var circles = [{"x": 10, "y": 30}, {"x": 10, "y": 60}, {"x": 10, "y": 90}];

svg.selectAll()
   .data( circles )
   .enter()
   .append("circle")
   .attr({
      r : 10;
      x : function(currentObject) { return currentObject.x; },
      y : function(currentObject) { return currentObject.y; }
   });
```

Trois cercles sont ainsi créés.

## 12.4 Mise en œuvre d'écouteurs d'événements

La méthode **on()** permet d'attacher des écouteurs d'événements à un ou plusieurs élements du DOM.
Voici quelques schémas de programmation ulilisant cette méthode :

```
<élément(s) du DOM> .on("click",       function() { ... })
                    .on("mouseover",   function() { ... })
                    .on("mouseout",    function() { ... });
```

Voici le même exemple de remplacement de données vu dans le paragraphe précédent mais maintenant déclenché par un clic sur l'un des textes :

```
let data = ["texte1", "texte2", "texte3"];

function update() {
   selection = d3.select("body").selectAll("h3").data(data);
   selection.exit().remove();
   selection.text( function(d) {return d; });
}

d3.select("body")
  .selectAll("h3")
  .data(data)
  .enter()
  .append("h3")
  .text( function(d) {return d;})
  .on("click", function() { update(); });

data = ["texte4", "texte5", "texte6"];
```

## 12.5 Production d'éléments graphiques sans association préalable avec des éléments du DOM :

Il est bien sûr aussi possible de créer des éléments graphiques via D3.js sans association avec des éléments du DOM.
Voici un exemple de production de courbes de ventes mensuelles :

```
var ventes = [{"produit1" : [150, 125, 83, 75, 40, 72, 88, 102, 110, 153, 170, 184]},
              {"produit2" : [67, 80, 94, 102, 130, 144, 106, 95, 65, 40, 51, 65]}];

$().ready(function() { // en utilisant JQuery (ce qui n'est pas obligatoire)
    d3.select("body").append("svg").attr("width", 700).attr("height", 700);
    for (var i in ventes) {
         var d = "";
         var abs = 50;
         for (var p in ventes[i]) { // un peu moche car il y a une seule propriété
                                    // possibilité d'utiliser Object.keys()
             d += "M"+abs+","+ventes[i][p][0];
             for (var j=1; j < 12; j++) {
                  abs += 50;
                  d += " "+abs+","+ventes[i][p][j];
             }
         }
        d3.select("svg").append("path").attr("d", d).attr("stroke", "black").attr("fill", "none");
    }
})
```

# 13 Créer une architecture clients-serveur (avec Node.js) :

Cette section montre l'utilisation des **web sockets** pour permettre à un client d'envoyer un message vers un serveur qui va lui-même renvoyer cette information à tous les autres clients.
Typiquement cette architecture est applicable à un jeu multi-joueurs ou à un chat. Les messages sont formatés en JSON.

Le serveur est construit grâce à **Node.js** une plateforme logicielle en Javascript permettant notamment de créer des serveurs.
Pour installer cette plateforme sous Linux : **sudo apt install nodejs**.
Cette plateforme étant modulaire, le serveur utilisera les modules nécessaires en les important.

## 13.1 Mise en place des web sockets :

Les codes nécessaires aux web sockets sont accessibles :

— pour le serveur :
en important **le module de Node.js** gérant les web sockets : **require("socket.io")**

— pour le client :
en important la bibliothèque **socket.io.js** → `https://github.com/socketio/socket.io-client`

La communication entre les clients et le serveur est initialisée :

— sur le serveur :
via la **méthode on() de l'objet io.sockets** invoquée sur le **message connection**
`io.sockets.on('connection', function (socket) { ... });`
La fonction de callback étant appelée avec un objet socket utilisé pour la réception de messages et l'envoi de messages au client courant.

— sur le client :
en créant **l'objet socket** :
`socket = io('http://localhost:8888');`

Les messages sont envoyés via les méthodes **io.emit()** ou **socket.emit()** :

— `socket.emit()` : envoi d'un message au serveur ou, de la part du serveur, au client courant ;
— `socket.broadcast.emit(<labelDuMessage>, objetMessage)` : envoi d'un message du serveur à tous les autres cleints que celui qui vient d'envoyer le message ; ;
— `io.emit(<labelDuMessage>, objetMessage)` : envoi d'un message à tous les clients.

Exemple d'émission par le serveur à tous les clients d'un objet contenant le numéro et le nom d'un nouveau joueur :
`io.emit('nouveauJoueur', {"numJoueur":nbJoueursConnectes, "nomJoueur":nomJoueur});`

Les messages sont reçus via **la méthode on() de l'objet socket** :
`socket.on(<labelDuMessage>, function(message) { ... });`

## 13.2 Le serveur (serveur.js) :

Votre application serveur est exécutée comme suit : **node serveur.js** ou **nodejs serveur.js**.
Dans l'exemple donné ci-après, le serveur écoute sur le port 8888.

Le serveur assure les transactions suivantes :
— sur un message *etat* le serveur renvoie au client des informations sur les joueurs déjà connectés ;
— sur un message *rejoindre* le serveur renvoie à tous les clients (et y compris à l'expéditeur du message) des informations sur le joueur qui rejoint la partie ;
— sur un message *quitter* le serveur renvoie à tous les clients (et y compris à l'expéditeur du message) des informations sur le joueur qui quitte la partie.

```
var nomsJoueurs = [];
var nbJoueursConnectes = 0;

var app = require('http').createServer(function(req, res){ res.end(html); });
app.listen(8888);
var io = require("socket.io").listen(app);

io.sockets.on('connection', function (socket) {

    socket.on('etat', function(message) {
              console.log("Etat d'une partie");
              var etat = {"nbJoueursConnectes":nbJoueursConnectes, "nomsJoueurs":nomsJoueurs};
              socket.emit('etat', etat);
    });

    socket.on('rejoindre',function(message) {
              nomJoueur = message["nomJoueur"];
              console.log("Nouveau joueur : "+nomJoueur);
              nomsJoueurs.push(nomJoueur);
              io.emit('nouveauJoueur', {"numJoueur":nbJoueursConnectes, "nomJoueur":nomJoueur });
              nbJoueursConnectes++;
    });

    socket.on('quitter',function(message) {
              numJoueur = message["numJoueur"];
              console.log("Ancien joueur : "+numJoueur);
              nomsJoueurs.splice(numJoueur, 1);
              nbJoueursConnectes--;
              io.emit('ancienJoueur', message);
    });
});
```

Vous remarquerez que le serveur "tourne" sur le port 8888.

## 13.3 Un client (client.html) :

Quand un internaute/client se connecte au serveur, celui-ci :
— lui renvoie la liste des joueurs connectés ;
— renvoie aux autres clients déjà connectés son pseudo.

Il faudrait rendre générique le nombre de joueurs connectés (ici maladroitement limité à quatre)...

```
<html>
  <head>
     <title> Client web socket </title>
     <script src="socket.io-client-master/socket.io.js"></script>
     <script>
        var socket;
        var nbJoueursConnectes = 0;
        var nomsJoueurs = [];
        var joueurLocal = -1;      // indice dans nomsJoueurs
```

```
        function rejoindrePartie() {
           if (joueurLocal == -1) {
               nomJoueur = document.getElementsByName('joueur')[0].value;
               if (nbJoueursConnectes < 4) {
                  if (nomJoueur != "") {
                      console.log("Envoi de la connexion");
                      socket.emit("rejoindre", { "nomJoueur": nomJoueur });
                      joueurLocal = nbJoueursConnectes;
                  }
               }
               else console.log("Vous ne pouvez pas pour l'instant rejoindre le groupe !");
           }
        }

        function quitterPartie() {
           if (joueurLocal > -1) {
               console.log("Suppression du joueur n."+joueurLocal);
               socket.emit("quitter", { "numJoueur": joueurLocal, "raison": "bye bye" } );
            }
        }

        function byebye(ancienJoueur) {
           var nomJoueur = nomsJoueurs[ancienJoueur];
           console.log("Du serveur ancienJoueur ="+ancienJoueur+"/"+nomJoueur);
           if (joueurLocal == ancienJoueur) {
               joueurLocal = -1;
               document.getElementsByName("joueur")[0].value = "";
           }
           else joueurLocal--;
           nomsJoueurs.splice(ancienJoueur, 1);
           nbJoueursConnectes--;
           for (var i=0; i < nbJoueursConnectes; i++)
                document.getElementById("joueur"+i).innerHTML = nomsJoueurs[i];
           document.getElementById("joueur"+i).innerHTML = "";
        }

        socket = io('http://localhost:8888');
        socket.emit("etat",{});  // Pour que le serveur renvoie les noms des joueurs déjà connectés

        socket.on("etat", function(data) {
                  console.log("Dans la réception d'état");
                  for (var m in data) {
                       console.log(m+" : "+data[m]);
                       window[m] = data[m];  // MAGIQUE
                       for (var i=0; i < nomsJoueurs.length; i++) {
                            console.log("joueur ="+nomsJoueurs[i]);
                            document.getElementById("joueur"+i).innerHTML = nomsJoueurs[i];
                       }
                  }
        });

        socket.on("nouveauJoueur", function(data) {
                  console.log("Du serveur : nouveau joueur");
                  nomsJoueurs.push(data["nomJoueur"]);
                  document.getElementById("joueur"+nbJoueursConnectes).innerHTML = data["nomJoueur"];
                  nbJoueursConnectes++;
        });

        socket.on("ancienJoueur", function(data) {
                  var ancienJoueur = data["numJoueur"];
                  byebye(ancienJoueur);
        });
    </script>
  </head>
```

```
  <body>
        Rejoindre la partie <input type="text" name="joueur"> </input>
        <button onClick="rejoindrePartie()"> Hello </button>
        <br/><br/>
        Quitter les vivants <button onClick="quitterPartie()"> Bye Bye </button><br/><br/>
        <br/><br/>
        <li> Joueur 1 : <label id="joueur0"> </label> </li>
        <li> Joueur 2 : <label id="joueur1"> </label> </li>
        <li> Joueur 3 : <label id="joueur2"> </label> </li>
        <li> Joueur 4 : <label id="joueur3"> </label> </li>
  </body>
</html>
```

# 14 Introduction au responsive web avec Bootstrap :

Le but premier de la bibliothèque **Bootstrap** est d'adapter l'interface de votre page web à l'écran de votre client : cette adaptation se nomme le **responsive web**.

Le système d'interfaçage le plus classique est l'utilisation d'une grille (*grid*).
Chaque ligne de cette grille est calibrée pour être divisée au maximum en 12 éléments.
Chaque élément d'une ligne de la grille (une colonne) doit indiquer combien d'éléments parmi les 12 potentiels sont recouverts.

Par ailleurs, la disposition que vous choisissez est calibrée pour un type d'écran sélectionné parmi les quatre suivants :
— pour un téléphone : code *xs*
— pour une tablette : *sm*
— pour un ordinateur à écran médian : *md*
— pour un ordinateur à écran large : *lg*

Si le client qui visualise votre page possède un écran plus petit que votre écran de référence, les éléments de la grille seront automatiquement disposés par Bootstrap pour qu'ils soient correctement affichés.

Voici l'exemple d'une grille de deux lignes calibrée pour un ordinateur à écran médian :
— la première ligne comportera trois éléments recouvrant un tiers de la ligne (donc 4 des éléments de base) ;

— la seconde ligne comportera deux éléments, le premier recouvrant un tiers de la ligne, le second les deux-tiers (donc 8 éléments de base).

Les URLs d'importation de la bibliothèque Bootstrap, des feuilles de style associées et de JQuery sont celles affichées sur la documentation officielle à l'heure où j'écris ce poly...

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Bootstrap Example</title>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.0/jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
</head>

<body>

<div class="row">
  <div class="col-md-4"> un tiers </div>
  <div class="col-md-4"> un tiers </div>
  <div class="col-md-4"> un tiers </div>
</div>
<div class="row">
  <div class="col-md-4"> un tiers </div>
  <div class="col-md-8"> deux tiers </div>
</div>

</body>
</html>
```

# Index