A la rencontre de la programmation objets et fonctionnelle en javascript
-
Gestion des événements
Accès au DOM
Chargement asynchrone de données (AJAX)
Interfaçage de (Google|OpenStreet) Maps
Création d'interfaces graphiques
Gestion de web sockets

Pierre Pompidor

Table des matières

1 Introduction :

1.1 Panorama de l'utilisation de Javascript :

Comme **technologie client(e)** (mise en oeuvre dans le navigateur), Javascript est utilisé :
— pour gérer l'événementiel lié à une page HTML, comme par exemple pour :
  — contrôler la soumission d'un formulaire ;
  — faire apparaître des popups ;

— pour accéder et modifier les éléments du **DOM (Document Object Model)**
  — avec l'API DOM de Javascript ;
  — avec l'utilisation de la bibliothèque **JQuery** ;

— pour gérer des flux de données asynchrones avec le serveur → **AJAX (Asynchronous Javascript and XML)** (toujours principalement avec l'utilisation de la bibliothèque **JQuery**)

— pour créer des interfaces graphiques :
  — en utilisant la balise `<canvas>` ;
  — via la bibliothèque **D3** qui interface **le langage SVG (Scalable Vector Graphics)**

— pour mettre en oeuvre les différentes facettes d'**HTML5**, comme par exemple :
  — en gérant le stockage de données sur le système de fichiers local (*localStorage...*)
  — en gérant des **web sockets** pour faire du push entre un serveur et des clients

*Javascript est également devenu central dans le développement web en proposant des architectures **client-serveur** dérivées du patron de conception MVC et reposant :*
— *au niveau du client (client-side), sur de nombreux frameworks dont les plus connus à ce jour sont :*
  — *AngularJS → https://docs.angularjs.org/api*
  — ***Angular 2 (avec TypeScript)** → https://angular.io/docs/ts/latest/index.html*
  — *ExtJS → http://docs.sencha.com/extjs/6.0/*
  — *Knockout → http://knockoutjs.com/documentation/introduction.html*
  — *Backbone → http://cq5-dev.com/wp-content/uploads/2013/05/backbonetutorials.pdf*
  — *Ember → http://vfsvp.fr/article/une-introduction-en-profondeur-a-emberjs/*
  — *React → https://facebook.github.io/react/docs/getting-started.html*

— *au niveau du serveur (server-side), sur **Node**, un serveur réputé pour son efficacité.*
*et par exemple avec l'**architecture MEAN** : MongoBD (SGBD NoSql) + Express + Angular + Node.*

Le framework **Angular 2** est enseigné dans le module de première année de Master intitulé *Présentation des données du web.*

Dans ce support de cours, un premier chapitre introduira à la programmation par objets (par prototypes, les classes n'existant pas nativement) en Javascript, tandis qu'un second s'intéressera plus particulièrement aux aspects de la programmation fonctionnelle même si les deux aspects (programmation objets et programmation fonctionnelle) sont complètement imbriqués en Javascript.)
Puis seront abordés, la gestion des événements, l'accès au DOM, le chargement asynchrone de données (AJAX) ainsi qu'un aperçu de l'interfaçage de (Google|OpenStreet) Maps, de la création d'interfaces graphiques et enfin la gestion de web sockets.

1.2 Où coder du code javascript ?

Pour une utilisation côté client, 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 :

test.html :
```
<html>
   <head>
      <script src="test.js" type="text/javascript" language="javascript"></script>
   </head>
   <body> Test de code Javascript </body>
</html>
```

test.js :
```
console.log("Bonjour !");
```

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

Il faut remarquer que le code Javascript **est directement exécuté** lors du chargement de la page web et peut donc l'être **avant que les éléments de la page (balises HTML...) ne soient instanciés dans l'arbre DOM** (qui est la représentation interne de la page web en mémoire du navigateur).
Il est donc souvent névessaire de différer cette exécution. Deux principales solutions existent (qu seront vues plus loin) :

- en javascript "pur", par l'écouteur d'événements `onLoad` associé à la balise `<body>`
- via la bibliothèque JQuery, par la méthode `ready` → `$().ready(function() { ... });`

Cela-dit, quelquefois dans les exemples listés ci-après, et par simplicité, les fonctions javascript seront définies dans des blocs `<script> ... </script>` directement inclus dans la partie d'entête du code HTML (mais il serait donc mieux de les définir dans des codes javascript externalisés).

1.3 Outils du navigateur

Les outils du navigateurs (`<F12>` pour les faire apparaître) vous permettent (entre autres) :

- **par la console** de visualiser les messages émis :
  - par l'interpréteur javascript (notamment les erreurs de programmation → liens cliquables) ;
  - par vos codes ;
- **par l'inspecteur DOM** d'afficher la hiérarchie des éléments de la page web.

Certains navigateurs proposent deux consoles :

- la console web → pour l'onglet courant ;
- la console du navigateur → pour tous les onglets.

2 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 est un ensemble de normes de programmation. Dans ce polycopié, nous ne détaillerons pas la grammaire complète du langage, mais il faut retenir les points listés dans les deux paragraphes suivants.

2.1 Variables :

— les variables sont **typées dynamiquement** (et donc pas lors de leurs déclarations) ;
— sont déclarées par :
  — le spécificateur **var** : de portée la fonction dans laquelle la variable apparaît (mais pas les fonction incluses) ;
  — le spécificateur **let** : de portée le bloc d'instructions ;
  — le spécificateur **const** : la variable n'est accessible qu'en lecture.

2.1.1 Types internes : six types primitifs et le type Object :

Javascript va typer (en interne) les variables suivant six types primitifs :
— type booléen : true et false ;
— type nul : null ;
— type indéfini (une variable n'existe pas ou n'a pas de valeur) : **undefined** ;
  exemple de variable créée mais de valeur indéfinie : **var i ;**
— type pour les nombres ;
— type pour les chaînes ;
— type pour les symboles ;

et le type **Object** pour tous les autres cas de figures...

2.1.2 Exemples sur la portée de variables créées par le spécificateur var :

```
<script>
    var i = 0;
    function fonction1() {
       console.log("i = "+i);  // i = 0
       function fonction2() {
          console.log("i = "+i);  // i = undefined
          var i = 'a';
          console.log("i = "+i);  // i = a
       }
       fonction2();
       console.log("i = "+i);  // i = 0
    }
    fonction1();
</script>
```

2.2 Structures itératives :

Outre les structures itératives :
— **while (...) {...}**
— **do {...} while (...) ;**
— **for (... ; ... ; ...) {...}**

Javascript propose :

→ la structure **for (... in ...) {...}** qui permet aussi bien d'itérer sur les propriétés d'un objet (les objets seront expliqués plus en détails dans la section suivante) ou les éléments d'une liste.
Par ailleurs il existe la structure **for (... of ...) {...}** qui pourra être aussi utilisée sur une liste.

→ la méthode **forEach(function(elt){...})** qui sera appliquée sur un objet liste (Array).

2.2.1 Structures itératives classiques (par l'exemple) :

```
<script>
        var liste = ['v1', 'v2'];

        var i = 0;
        while ( i < liste.length ) {
            console.log(i+" : "+liste[i]);
            i++;
        }

        i = 0;
        do {
            console.log(i+" : "+liste[i]);
        } while ( ++i < liste.length );

        for (let i=0; i < liste.length; i++) {
            console.log(i+" : "+liste[i]);
        }
</script>
```

2.2.2 Structure for (... in ...) :

La notation "littérale" des objets sera expliquée dans la section suivante.

```
<script>
        var liste = ['v1', 'v2'];
        for (let i in liste) {
            console.log(i +" : "+liste[i]);  // 0 : v1
                                             // 1 : v2
        }

        var objet = {'p1' : 'v1', 'p2' : 'v2'};
        for (let p in objet) {
            console.log(p +" : "+objet[p]);  // p1 : v1
                                             // p2 : v2
        }
</script>
```

2.2.3 Structure for (... of ...) :

La structure **for (let i in ...) {...}** sur une liste instancie dans la variable les indices des éléments de la liste, pour extraire directement leurs valeurs il faut utiliser la structure **for (... of ...) {...}** :

```
<script>
        var liste = ['v1', 'v2'];
        for (let v of liste) {
            console.log(v);  // v1
                             // v2
        }
</script>
```

2.2.4 Méthode forEach pour les listes :

```
<script>
        var liste = ['v1', 'v2'];
        liste.forEach(function(element) {
            console.log(element); // v1
                                  // v2
        });
</script>
```

3 Introduction à la programmation objets avec Javascript :

Javascript est donc une implémentation de la norme **ECMAScript** (ActionScript en est par exemple une autre).

Un langage objets :
Si quasiment tout est objet en Javascript, **il n'y a pas de classes** contrairement à JAVA ou C++, l'héritage étant assurée par des objets particuliers appelés **prototypes** : ce type d'héritage est appelé **héritage par délégation**.

Un langage fonctionnel : :
Les fonctions (méthodes...) ont la primauté en Javascript, ce sont les éléments du langage qui sont mis en avant-plan.

Ainsi nous pourrons :
— **(En)chaîner les méthodes** → utilisation massive dans l'API JQUERY ;
— **Passer une fonction en paramètre à une fonction** → fonctions de **callback** ;
— **Retourner une fonction dans une fonction** → par exemple réalisation de "Factory" de fonctions.

3.1 Création directe d'un objet en Javascript / JSON :

Contrairement à de nombreux autres langages de programmation, les objets peuvent être créés littéralement en Javascript (**et ne sont donc 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 celle qui est la plus populaire dans l'échange de données (devant XML)).

(*) Pour gérer des classes en Javascript, il existe des extensions "de plus haut niveau" :
— **TypeScript** → `http://www.typescriptlang.org/`
— **Dart** → `https://www.dartlang.org/`

Dans cet exemple, un objet *DemisRoussos* est défini :

```
var DemisRoussos = { nom : "Roussos",
                     prenom : "Artémios"};
```

Il est important de remarquer qu'une propriété d'un objet peut avoir comme valeur :
— une valeur scalaire (entier, réel, chaîne de caractères...)
— une liste (tableau dynamique), par exemple `['canard', 'oie', 'poule']`
— un objet
— voire le code implémentant une fonction...

Il est aussi intéressant de noter que :
— même si formellement 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*.

3.2 Héritage par chaînage de prototypes :

Outre les deux propriétés `nom` et `prenom`, l'objet `DemisRoussos` va contenir la propriété interne `__proto__` qui est la référence à un autre objet appelé le **prototype** de cet objet (*) et duquel il peut hériter (voir paragraphe suivant).
Par défaut cet objet contiendra lui-même une propriété `__proto__` de valeur null.

(*) En fait, comme il le sera expliqué dans le paragraphe suivant, l'objet chaîné doit offrir en héritage **un sous-objet nommé prototype**.

Cet objet prototype va permettre de définir les propriétés et les méthodes héritées :

```
var chanteurDeLegende = {quiSuisJe : function() {
    console.log("Je suis "+this.prenom+" "+this.nom+", un chanteur de légende !"); }
                        };

var DemisRoussos = { nom : "Roussos",
                     prenom : "Artémios",
                     __proto__ : chanteurDeLegende};

DemisRoussos.quiSuisJe();
```

Vous remarquerez que **this** désigne l'objet originel (appelé aussi **l'objet appelant** (le calling object)) passé à l'objet chaîné (`chanteurDeLegende`).

3.3 La propriété prototype de l'objet hérité :

De manière conventionnelle (mais l'opérateur new vu ci-après adoptera ce comportement ;)), les propriétés héritées doivent être regroupées dans un objet appelé **prototype** :

```
var chanteurDeLegende = {prototype : {quiSuisJe : function() {
                                       console.log("Je suis "+this.prenom+" "+this.nom+", ..."); }
                                      }
                         };

var DemisRoussos = { nom : "Roussos",
                     prenom : "Artémios",
                     __proto__ : chanteurDeLegende.prototype};
DemisRoussos.quiSuisJe();
```

3.4 Création d'un objet par appel d'une fonction constructeur :

Un moyen plus commun (et plus pratique ;)) de créer un objet est d'utiliser l'opérateur `new` qui crée un objet à partir d'une fonction constructeur (on devrait dire constructrice ;)).

Dans cet exemple, les objets *DemisRoussos* et *Christophe* sont créés par la fonction constructeur *ChenteurDeLegende*. Dans le code de cette fonction, **this** représente l'objet en cours de création et c'est l'opérateur **new** qui réalise automatiquement le chaînage de la propriété `prototype` sur la propriété `__proto__` de l'objet appelant.

```
function chanteurDeLegende(nom, prenom) {
   this.nom = nom;
   this.prenom = prenom;
}
chanteurDeLegende.prototype.quiSuisJe = function() {
   console.log("Je suis "+this.prenom+" "+this.nom+", un chanteur de légende !");
}

var DemisRoussos = new chanteurDeLegende("Roussos", "Artémios");
var Christophe   = new chanteurDeLegende("Bevilacqua", "Daniel");
DemisRoussos.quiSuisJe();
Christophe.quiSuisJe();
```

ou encore en utilisant une fonction externalisée :

```
var quiSuisJeExterne = function() { console.log("Je suis "+this.prenom+" "+this.nom+", ..."); }
function chanteurDeLegende(nom, prenom) {
   this.nom = nom;
   this.prenom = prenom;
}
chanteurDeLegende.prototype.quiSuisJe = quiSuisJeExterne;

var DemisRoussos = new chanteurDeLegende("Roussos", "Artémios");
DemisRoussos.quiSuisJe();
```

Vous remarquerez que sans utiliser la référence à l'objet appelant (`this`), on ne pourrait pas accéder à la méthode `quiSuisJe()` (sa référence n'étant pas copiée dans la propriété `quiSuisJe` de l'objet originel).

Ainsi, le code suivant ne fonctionne pas :

```
function chanteurDeLegende(nom, prenom) {
   this.nom = nom;
   this.prenom = prenom
   var quiSuisje = function() {
       alert("Je suis "+this.prenom+" "+this.nom", un chanteur de légende");}
}
var DemisRoussos = new chanteurDeLegende("Roussos", "Artémios");
Demis_Roussos.quiSuisJe();
```

Dans cette nouvelle version, la fonction `quiSuisJe()` est délocalisée dans le prototype.
La propriété **prototype** d'une fonction (qui en interne est elle-même gérée comme un objet) référence un objet dont les propriétés pourront être héritées par tout objet qui prendra le fonction comme modèle.

```
function chanteurDeLegende(nom, prenom) {
   this.nom = nom;
   this.prenom = prenom;}

chanteurDeLegende.prototype.quiSuisJe = function() {
   console.log("Je suis "+this.prenom+" "+this.nom+", un chanteur de légende !");}

var DemisRoussos = new chanteurDeLegende("Roussos", "Artémios");
var Christophe   = new chanteurDeLegende("Bevilacqua", "Daniel");

DemisRoussos.quiSuisJe();
Christophe.quiSuisJe();
```

3.5 Itérer sur les propriétés chaînées d'un objet :

Pour connaître les propriétés (avec y compris celles chaînées ou pas) plusieurs méthodes nous sont offertes :

```
var objetProto = { proprietePere:"pere",
                   prototype: {proprieteHeritable : "heritage" }};

var objetDerive = { proprieteFils:"fils",
                    __proto__ : objetProto.prototype};

for (var p in objetDerive) {
     console.log(p+" : "+objetDerive[p]);  // proprieteFils : fils
                                           // proprieteHeritable : heritage
}

var proprietes = Object.keys(objetDerive);
for (var i in proprietes) {
     console.log(proprietes[i]+" : "+objetDerive[proprietes[i]]);  // proprieteFils : fils
}

proprietes = Object.getOwnPropertyNames(objetDerive)
for (var i in proprietes) {
     console.log(proprietes[i]+" : "+objetDerive[proprietes[i]]);  // proprieteFils : fils
}
```

La différence d'emploi entre Object.keys(...) et Object.getOwnPropertyNames(...) est particulièrement subtile : Object.keys(...) ne permet pas d'énumérer les propriétés non énumérables (par exemple la propriété length des listes...).

3.6 Là où les choses se compliquent : l'héritage

Etudions un exemple plus complexe.
Essayons de reproduire ce que nous pourrions faire dans un langage objets à classes : dériver une classe fille à partir d'une classe mère ! Par exemple que les chanteurs de légendes "instances" de l'objet `chanteurDeLegende` :

— qui ont un `pseudo` intemporel

héritent des propriétés et des méthodes définies dans un objet `chanteur` :

— `prenom` et `nom` ;
— `quiSuisJe()`.

Pour cela, une première méthode (ce n'est pas la seule...) consiste à :
— donner comme prototype de *l'objet "fils"* celui de *l'objet "père"* ;
— utiliser la fonction **call** (propriété de l'objet-fonction) qui permet de relayer la référence de l'objet appelant à la fonction de niveau supérieur utilisée comme modèle d'héritage (dans l'exemple suivant `chanteurDeLegende`).

Nous commençons par créer l'objet "père" dont les propriétés seront héritées par l'objet "fils" `chanteurDeLegende` :

```
function chanteur(prenom, nom) {
   this.prenom = prenom;
   this.nom = nom;}

chanteur.prototype.quiSuisJe = function() {
   console.log("Je suis "+this.prenom+" "+this.nom+" de pseudo "+this.pseudo+", ...");
}

function chanteurDeLegende(nom, prenom, pseudo) {
   chanteur.call(this, prenom, nom); // et non chanteur(prenom, nom);
   this.pseudo = pseudo;
}
chanteurDeLegende.prototype = new chanteur();

var DemisRoussos = new chanteurDeLegende("Roussos", "Artémios", "LeGrecChantant");
var Christophe   = new chanteurDeLegende("Bevilacqua", "Daniel", "Christophe");

DemisRoussos.quiSuisJe();
Christophe.quiSuisJe();
```

Remarquez que si vous remplacez l'appel à `call()` par un appel direct de fonction, la méthode `quiSuisJe()` est bien héritée grâce au chaînage de prototypes, mais seule la propriété `pseudo` sera reconnue, car la valeur de `this` a été perdue lors de l'appel direct de `chanteur` (qui crée grâce à `this` les propriétés `nom` et `prenom` dans l'objet appelant (alors que `this` serait bien positionnée par l'appel d'un modèle avec `new`))) *(cette phrase est sans doute un peu difficile à comprendre ;))* .

Exercice :

Suite à l'exercice précédent, modélisez et implémentez le fait qu'un chanteur puisse être aussi auteur et/ou compositeur :
— si un chanteur est auteur, il "héritera" d'une propriété qui listera les chansons qu'il a écrites ;
— si un chanteur est compositeur, il "héritera" d'une propriété qui listera les musiques composées.

Extrait de la documentation officielle sur la fonction `call()` : :

```
You can assign a different this object when calling an existing function. this refers to the
current object, the calling object.
With call, you can write a method once and then inherit it in another object, without having
to rewrite the method for the new object.
Examples : Using call to chain constructors for an object

You can use call to chain constructors for an object, similar to Java. In the following example,
the constructor for the Product object is defined with two parameters, name and value.
Two other functions Food and Toy invoke Product passing this and name and value.
Product initializes the properties name and price, both specialized functions define the category.
```

```
function Product(name, price) {
   this.name = name;
   this.price = price;

   if (price < 0)
     throw RangeError('Cannot create product "' + name + '" with a negative price');
   return this;
 }

function Food(name, price) {
  Product.call(this, name, price);
  this.category = 'food';
}
Food.prototype = new Product();

function Toy(name, price) {
  Product.call(this, name, price);
  this.category = 'toy';
}
Toy.prototype = new Product();

var cheese = new Food('feta', 5);
var fun = new Toy('robot', 40);
```

3.7 Un autre aspect de la programmation par objets : le chaînage de méthodes

Dans beaucoup de frameworks (comme `JQuery`), des méthodes peuvent être enchaînées comme suit : `objet.fonction1().fonction2().....fonctionn();`

Mais comment implémenter cela ? → étudions un exemple :

```
function ajoute(chaine) {  this.valeur += chaine; return this; }

function creeChaine(chaine) { this.valeur = chaine; }
creeChaine.prototype.ajoute = ajoute;

var chaineInitiale = new creeChaine("2");
console.log(chaineInitiale.ajoute('Fast').ajoute('4U').valeur);
```

4 La programmation fonctionnelle en Javascript :

En programmation fonctionnelle → **les fonctions sont des éléments de premier plan**, et quelque fois cela peut-être assez déroutant...

4.1 Passer une fonction en paramètre à une fonction → réalisation de fonctions de callback

Une fonction peut accepter en paramètre une fonction. Voyons un exemple :

```
function chercheAdresse(adresse, callback) {
     var expression = /Montpellier/i;
     if (expression.test(adresse)) {
         callback(1, {latitude  : 43.6,
                      longitude : 3.9})
     }
     else {
         callback(0, {});
     }
}

chercheAdresse("1 rue des rêves, 34000 Montpellier",
     function(status, resultat){
         if (status) {
               alert( "Latitude : "+resultat.latitude
                     +" longitude : "+resultat.longitude);
         }
         else {
               alert("Je ne connais que Montpellier !");
         }
     }
)
```

Une **fonction de callback** (dite encore fonction de rappel) est une fonction :
— qui va être généralement appelée quand se terminera la fonction à laquelle elle est passée en paramètre
— qui peut être codée par le programmeur qui utilise la bibliothèque contenant la fonction à laquelle elle est passée en paramètre...

4.1.1 Exercice :

Une autre facette de `JQuery` est de permettre d'appliquer une fonction sur les éléments d'une liste (ou les propriétés d'un objet) via la méthode `each` :
Exemple : `liste.each(fonction à exécuter);`

Soit le programme suivant (qui dévoile au passage comment passer un nombre quelconque de paramètres à une fonction javascript) :

```
function creeListe() {
    this.chaines = new Array();
    for(var i=0; i<arguments.length; i++) {
        this.chaines.push(arguments[i]);
    }
}
var chaines = new creeListe("chaine1", "chaine2", "chaine3");
```

Complétez ce programme pour :
— créer une fonction `each` la plus générique possible qui puisse appliquer une fonction quelconque sur la liste de chaînes de caractères ;
— appelez-là dans le but de faire afficher successivement la valeur de chaque chaîne :
chaines.each(...) ;

4.2 Retourner une fonction dans une fonction→ réalisation de "Factories" de fonctions

Une fonction peut retourner une fonction. Voyons un exemple :

```
var chanteur = function(prenom, nom) {
    this.prenom = prenom;
    this.nom = nom;
    this.albums = new Array();

    this.listeAlbums = function () {
        for (var num_album in this.albums) {
            console.log("Album : "+this.albums[num_album]);
        }
    }
}

var gestionAlbums = function (chanteur) {
    return function set(nomAlbum) {
        chanteur.albums.push(nomAlbum);
    };
};

var demisRoussos = new chanteur("Artemios", "Roussos");
demisRoussos.albums.push("The golden voice");
demisRoussos.listeAlbums();

var gestionAlbumsDemisRoussos = gestionAlbums(demisRoussos);
gestionAlbumsDemisRoussos("Forever and ever");
demisRoussos.listeAlbums();
```

La fonction `set` qui est appelée lors de l'invocation de la nouvelle fonction `gestionAlbumsDemisRoussos` peut toujours accèder au paramètre `chanteur` : c'est une **fermeture**.

Définition d'une fermeture (Wikipedia) : fermeture ou clôture (en anglais, closure) est une fonction qui capture des références à des variables libres dans l'environnement lexical. Une fermeture est donc créée, entre autres, lorsqu'une fonction est définie dans le corps d'une autre fonction et fait référence à des arguments ou des variables locales à la fonction dans laquelle elle est définie.

5 La gestion d'événements :

Le plus ancien usage de Javascript est de gérer au niveau du client (càd 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 (par exemple pour gérer des effets "artisanaux" de menus déroulants...)).

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>`) ;
- `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` :

```
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
      <style type="text/css">
         ...
      </style>
      <link rel="stylesheet" type="text/css" href="....css" />

      <script language="javascript" type="text/javascript">
        // code javascript global
        function fonctionJavascript (parametres) {
           ...
        }
        ...
      </script>
      <script language="javascript" type="text/javascript" src="....js"></script>
   </head>
```

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 navaigateur 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.

Une première possibilité d'interaction avec le DOM est de **sélectionner** des éléments de celui-ci

- via la référence à l'objet qui vient d'être sélectionné → **this**
- via différentes méthodes de l'objet document :
  - 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.

5.1 Exemples :

5.1.1 Changement de la taille d'une image lors d'un clic :

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

5.1.2 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>
```

6 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 éléments 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.

6.1 Avec l'API DOM de Javascript :

Voici quelques fonctions de l'API DOM de Javascript :

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

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

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

6.2 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" />
```

Des noeuds du DOM peuvent être sélectionnés `$(sélecteur)` en utilisant une syntaxe qui reprend celle de CSS. Voici les sélecteurs les plus courants :

```
balise
#id
.class
*
parent>enfant
:first
:last
:eq()
:type
:contains
élément[attribut=valeur]
```

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

6.2.1 Exemple :

Dans cet exemple :
— `ready()` n'exécutera le code JQuery qu'à partir du moment où le DOM 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");
});
```

6.3 Insertion de nouveaux noeuds dans le DOM :

JQuery nous propose toute une série de méthodes pour insérer un nouveau noeud 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');
```

6.3.1 Exemple :

Dans cet exemple, 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);
```

6.4 Manipulation des attributs et des valeurs d'un noeud du DOM :

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

6.5 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);
});
```

7 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.

7.1 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.

7.2 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" />
```

7.2.1 Ajax en JQuery :

JQuery propose trois fonctions de haut niveau pour opérer des téléchargements de données :
— `$.ajax()` : la fonction la plus générique ;
— `$.get()` : pour télécharger des données notamment en XML ;
— `$.getJSON()` : pour télécharger spécifiquement des données sérialisées en JSON.
Nous nous intéresserons

7.2.2 Téléchargement de données XML :

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

```
$.get('XML/data.xml', function(data) {
     data.find(...).each(function() {
          ...
     });
});
```

7.2.3 Téléchargement de données JSON :

Dans ce schéma de programmation :
— `entryIndex` correspond au numéro d'ordre de chaque objet de la collection
— `entry` correspond à chaque objet de la collection

```
$.getJSON('JSON/data.json", function(data) {
     $.each(data, function(entryIndex, entry) {
          ...
     });
});
```

8 Interfaçage de Maps :

8.1 Avec Google Maps :

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

```
<script type="text/javascript" src="http://maps.google.com/maps/api/js?sensor=false">
```

8.1.1 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>
```

8.1.2 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);
```

8.1.3 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: "JE SUIS ICI !",
                                      draggable: false
                                    });
```

L'affichage de la popup 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);
                              });
```

8.2 Avec OpenLayers et OpenStreetMap (OSM) :

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

La **bibliothèque OpenLayers** après téléchargement (`http://openlayers.org/download/`) se lie à votre code via la balise `<script>` :

```
<script src="v3.14.0/build/ol.js" type="text/javascript"></script>
```

8.2.1 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) :

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

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

8.2.2 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).

```
var map = new ol.Map({
    target: 'map',
    layers: [
          new ol.layer.Tile({
            source: new ol.source.MapQuest({layer: 'osm'}) // ou 'sat' ou 'hyb'
          })
        ],
    view: new ol.View({
          center: ol.proj.fromLonLat([3.87667,43.6111]),
          zoom: 15 // max 11 pour vue satellitaire (sat)
        })
});
```

8.2.3 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") };
```

9 Création d'interfaces graphiques :

Javascript peut manipuler la balise `<canvas>` qui offre un support à du dessin vectoriel.
Cela-dit, je vous recommande d'utiliser la blibliothèque **D3** → `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" />
```

9.1 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 `<circle r="..." x="..." y="..." />`) par rapport à une collection d'objets précisant leurs propriétés.

9.2 Exemple de code SVG "statique" :

Décrouvez le résultat dans votre navigateur...

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

<circle cx="70" cy="95" r="50" style="stroke:black;fill:none"/>
<circle cx="55" cy="80" r="5" style="stroke:black;fill:black"/>
<circle cx="85" cy="80" r="5" style="stroke:black;fill:black"/>
<g id="moustaches">
   <line x1="75" y1="95" x2="135" y2="85" style="stroke:black;"/>
   <line x1="75" y1="95" x2="135" y2="105" style="stroke:black;"/>
</g>
<use xlink:href="#moustaches" transform="scale(-1 1) translate(-140 0)" />
<polyline points="108 62, 90 10, 70 45, 50 10, 32 62" style="stroke:black; fill:none" />
</svg>
```

9.3 Création d'objets graphiques à partir d'une collection :

Dans cet exemple, les rayons des cercles sont fixés, mais leurs positionnements dépendent de la collection *cercles* :

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

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

9.4 Attachement d'écouteurs d'événements :

La méthode **on** permet d'attacher des écouteurs d'événements à un objet, exemples :

```
   ... .on("click",       function() { ... })
       .on("mouseover",   function() { ... })
       .on("mouseout",    function() { ... });
```

9.5 Sélection d'un élément :

La sélection d'élément peut être opérée par la fonction *select()* (elle pourrait aussi se faire via JQuery).

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

10 Créer une architecture clients-serveur :

Cette section montre l'utilisation des **web sockets** pour permettre à un client d'envoyer une information 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 tchat.

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

10.1 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 qui va être utilisé pour l'envoi et la réception de messages.

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

Les messages sont envoyés via différentes méthodes de l'**objet io**, par exemple **emit** :
`io.emit('labelDuMessage', objetMessage);`

Exemple de l'émission d'un objet contenant le numéro et le nom du dernier joueur connecté :
`io.emit('nouveauJoueur', {"numJoueur":nbJoueursConnectes, "nomJoueur":nomJoueur});`

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

10.2 Le serveur (serveur.js) :

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

```
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) {
    // Pour broadcaster : socket.broadcast.emit('monEvenement', mesDonnees);
    // Y compris avec l'appelant : io.emit()

    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);
    });
});
```

10.3 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="http://localhost/socket.io-client-master/socket.io.js"></script-->
     <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>
```