Cours 2
Javascript et Ajax

Yacine Bouzidi

Yacine.bouzidi@inria.fr

Plan

- Javascript
  - Objets
  - Portée des variables
  - Tableaux
  - DOM (Document object Model)
  - Evenements
- Ajax
  - XMLHttpRequest
  - Chargement dynamique de script

Javascript : objets

Programmation Orientée Objet

En général, un langage orienté objet statiquement typé propose une notion de classe et d'objet. Par exemple en Java :

```
class Point {
private int x;
private int y;
Point(int x, int y) {
        this.x = x;
        this.y = y;
}
public void move(int i, int j) {
        this.x += i;
        this.y += j;
}
public void getX() { return this.x; }
public void getY() { return this.y; }
}
```

Une classe définit un ensemble d'objets contenant un état interne (attributs : x, y) ainsi que du code (les méthodes : move, ...) permettant de manipuler cet état.
Un objet est l'instance d'une classe.

Concepts objets

Les langages orientés objets exposent généralement plusieurs concepts :

- La notion de constructeur : une fonction ayant un statut spécial, qui est appelé pour initialiser l'état interne de l'objet à sa création.
- Une notion de contrôle d'accès aux attributs et méthodes.
- Un moyen de référencer l'objet dans lequel on se trouve (this).
- Plusieurs notions permettant de partager du code ou des données (static, héritage, ...)

Objets en Javascript

Javascript ne fait pas de différences entre « attributs » et « méthodes ». Les « champs » d'un objet sont appelés «propriétés».

L'affectation à une propriété en Javascript ajoute celle-ci si elle est inexistante :

```
var p1 = { }; //Un objet vide
p1.x = 0; //On ajoute un champ x initialisé à 0
p1.y = 0; //On ajoute un champ y initialisé à 0

//Ajout de « méthodes » move, getX et getY

p1.move = function (i, j) { p1.x += i; p1.y += j; };
p1.getX = function () { return p1.x; };
p1.getY = function () { return p1.y; };
```

Problèmes avec le code ci-dessus

- Il faut copier-coller tout le code si on veut créer un autre point p2
- Pour chaque objet pi, on va allouer 3 fonctions différentes (qui font la même chose pour l'objet considéré.)

Première solution

```
var mkPoint = function (x, y) {
var p = { };
p.x = x;
p.y = y;
p.move = function (i, j) { p.x += i; p.y += j; };
p.getX = function () { return p.x; };
p.getY = function () { return p.y; };
return p;
};
…
var p1 = mkPoint(1,1);
var p2 = mkPoint(2, 10);
var p3 = mkPoint(3.14, -25e10);
…
```

La fonction mkPoint fonctionne comme un constructeur. Cependant, les trois «méthodes» sont allouées à chaque fois

Prototype de fonctions

Toute fonction déclaré possède nativement une propriété « prototype » initialement vide.

```
var Point = function(){}; // déclaration de  constructeur
Point.prototype; // Le prototype existe mais il est vide
```

Une propriete rajouté sur le prototype du constructeur devient disponible sur les instances

```
Point.prototype.move = function{...}; // cette méthode est
// definie sur le prototype
```

Function, prototype et new

En Javascript, le type «Function» (des fonctions) a un **statut particulier**.

Lorsque l'on appelle l'expression `new f(e1, …, en)` :

1. Un nouvel objet o (vide) est créé.
2. Le champ prototype de f est copié dans le champ prototype de o.
3. `f(e1, …, en)` est évalué et l'identifiant spécial this est associé à o
4. Si f renvoie un objet, alors cet objet est le résultat de `new f(e1, …, en)`, sinon l'objet o est renvoyé.

L'expression `new e` où e n'est pas un appel de fonction provoque une erreur.

Comment créer des objets avec ça ?

Function, prototype et new(suite)

```
var Point = function (x, y) {
this.x = x;
this.y = y;
};
…
var p1 = new Point(1,1);
var p2 = new Point(2, 10);
var p3 = new Point(3.14, -25e10);
…
```

1. Un nouvel objet p1 (vide) est créé.
2. Le champ prototype de Point est copié dans le champ prototype de p1.
3. `Point(e1, …, en)` est évalué et l'identifiant spécial this est associé à p1
4. Si `Point` renvoie un objet, cet objet est le résultat de `new Point(e1, …, en)`, sinon l'objet p1 est renvoyé.

Prototype et les propriétés propres

La résolution de propriété en Javascript suit l'algorithme ci-dessous. Pour rechercher la propriété p sur un objet o :

```
soit x ←o;
réptéter:
        si x.p est défini alors renvoyer x.p;
        si x.prototype est défini, et est un objet,
alors x ←x.prototype
```

Une propriété p d'un objet o peut donc être :

- Soit rattachée à o lui-même (on dit que p est une propriété propre de o, own property)
- Soit rattachée à l'objet o1 se trouvant dans le champ prototype de o (s'il existe)
- Soit rattachée à l'objet o2 se trouvant dans le champ prototype de o1 (s'il existe)
- ...

Function, prototype et new(fin)

```
var Point = function (x, y) {
        this.x = x;
        this.y = y;
};
Point.prototype.move = function (i, j) {
        this.x+= i;
        this.y+= j;
};
Point.prototype.getX = function () {
        return this.x;
};
Point.prototype.getY = function () {
        return this.y;
};
…
var p1 = new Point(1,1);
p1.move(2, 10);
…
```

Lors de l'appel à move, l'objet p1 ne possède pas directement de propriété move.

La propriété move est donc cherchée (et trouvée) dans son champ prototype.

Parallèle avec les langages compilés

- Le fonctionnement par prototype est identique à la manière dont les langages OO statiquement typés (Java, C++, C#) sont compilés.
- En Java, chaque objet contient un pointeur (caché) vers un descripteur de classe (une structure contenant les adresses de toutes les méthodes de la classe + un pointeur vers le descripteur de la classe parente) ≡ prototype.

Qu'offre Javascript en plus ?

- Redéfinition locale de propriétés (monkey patching)
- Définition manuelle du prototype pour « hériter » d'un type existant

Monkey patching

Technique qui consiste à redéfinir une méthode sur un objet spécifique (impossible à faire en Java).

```
var p1 = new Point(1, 1);
var p2 = new Point(1, 1);

p2.move = function () { this.x = 0; this.y = 0;};
p1.move(10, 10); //appelle Point.prototype.move
p2.move(10, 10); //appelle la méthode move définie ci-dessus

var x1 = p1.getX(); //x1 contient 11
var x2 = p2.getX(); //x2 contient 0
```

Technique dangereuse, car elle donne un comportement non uniforme à des objets de même « type ».

Différence entre propriété propre et prototype

On peut savoir à tout moment si un objet o a une propriété p propre en utilisant la méthode `.hasOwnProperty` (...)

```
var p = new Point(1, 2);
p.hasOwnProperty('x'); // renvoie true
p.hasOwnProperty('move'); // renvoie false
```

Héritage

L'algorithme de résolution de propriété peut être utilisé pour simuler l'héritage.

```
var ColoredPoint = function (x, y, c) {
        Point.call(this, x, y); //appel du constructeur parent
        this.color = c || "black"; //si c est convertible
//en false (en particulier undefined), on initialise à black
};

ColoredPoint.prototype = Object.create(Point.prototype);
//Object.create crée un nouvel objet dont le champ prototype
// est une copie de celui passé en argument.
ColoredPoint.prototype.getColor = function ()
                                  { return this.color; };
var p = new ColoredPoint(1, 2,"red");
p.move(10, 10); //se trouve dans ColoredPoint.prototype !
p.getColor(); //se trouve dans ColoredPoint.prototype !
```

Javascript : Portée des variables

Objet Global et variables globales

La norme Javascript (ECMA-262) définit un Objet Global initialisé avant le début du programme.

Les variables globales ne sont que des propriétés propres de cet objet.

Dans les navigateurs Web, cet objet global représente l'«onglet courant». Il possède une propriété window qui pointe sur lui même.

```
//On suppose que l'on est dans un fichier test.js
//inclus directement dans la page

var foo = 123; // variable « globale »
bar = 456; // assigne la propriété bar à l'objet global
this.baz = 789; // idem mais avec un this explicite

window.foo; // vaut 123
window.bar; // vaut 456
window.baz; // vaut 789
```

Variables locales

Une variable déclarée (au moyen de **var**) dans une fonction est locale à cette fonction.

Les « blocs » ne créent pas de nouvelle portée! :

```
//On suppose que l'on est dans un fichier test.js inclus direct
//dans la page
for(var i = 0; i < 10; i++) {
    var j = 2 * i;
    …
}
i; //bien défini après la boucle, vaut 9
j; //bien défini après la boucle, vaut 18
var f = function () {
    var k = 20;
    …
};
k; //vaut undefined
```

Shadowing

Une variable peut masquer une variable de même nom se trouvant dans une portée englobante :

```
//On suppose que l'on est dans un fichier test.js inclus
//directement dans la page
var x = 123;
console.log(x); // affiche 123

function f () {
    var x = 456;
    function g () {
        var x = 789;
        console.log(x); // affiche 789 quand g est appelée
    };
    g();
    console.log(x); // affiche 456 quand f est appelée
};
f();
console.log(x); // affiche 123
```

Identifiant this

L'identifiant this est similaire à celui de Java mais est tout le temps défini, avec des règles précises :

```
o.f(e1, …, en); //this est initialisé à o
new f(e1, …, en) : //initialisé à un objet fraîchement créé.
f.call(o,e1, …, en) : //initialisé à o.
f(e1, …, en) : //initialisé à l'objet global (window).
```

```
ColoredPoint.prototype.getColor = function () {
    console.log(this.color); //accès à la couleur
    var g = function () {
        console.log(this.color); //undefined car équivalent à
//window.color
    };
    g();
    return this.color;
};
```

Encapsulation

Javascript ne disposant ni de modules, ni de namespace, ni de classes, ni de packages, il est possible de rapidement «polluer» l'objet global :

```
//Définition de Point, move, getX, …
var Point = function (x, y) {
   …
};
…
var strAux (x, y) { //on définit une fonction auxiliaire
   return "(" + x + "," + y + ")";
};
Point.prototype.toString = function () {
   return strAux(x, y);
};
window.strAux (…); //est défini
strAux(…); //est défini
```

Problème : une fonction auxiliaire qui n'est utile qu'à la classe Point est visible globalement.

Utilisation de fonctions pour encapsuler

Comme les fonctions introduisent une nouvelle portée, on peut s'en servir pour encapsuler le code.

```
//Définition de Point, move, getX, …
var Point = function (x, y) {…};
Point.prototype = (function () {
//On est maintenant dans une fonction, strAux ne pourra pas
//s'échapper dans le contexte global
   var strAux (x, y) {
   return "(" + x + ", " + y + ")";
   };
//On renvoie un objet faisant office de prototype :
   return {
      move : function (i, j) { … },
      getX : function () { return this.x; },
      getY : function () { return this.y; },
      toString : function () { return strAux(x,y); }
   };
})(); //On applique immédiatement la fonction !

strAux(…); //undefined
```

Javascript : Tableaux

Array

Les tableaux (classe Array) font partie de la bibliothèque standard Javascript. On peut créer un tableau vide avec [ ].

```
var tab = [];
tab[35]; //undefined
tab[35] = "Hello"; //initialise la case 35 à "Hello"
tab[0]; //toujours undefined;
tab.length; //36 ! indice le plus grand ayant été
//initialisé + 1
```

Array

```
new Array(n) : //Initialise un tableau de taille n
//(indicé de 0 à n-1) où toutes les cases valent undefined
.length : //renvoie la longueur du tableau
.toString() : //applique .toString() à chaque élément
//et renvoie la concaténation
.push(e) : //ajoute un élément en fin de tableau
.pop() : //retire et renvoie le dernier élément du tableau.
//undefined si le tableau est vide
.shift() : //retire et renvoie le premier élément du tableau.
//undefined si le tableau est vide
.unshift(e) : //ajoute un élément au début du tableau
.splice(i, n, e1, …, ek) : //à partir de l'indice i, efface
//les éléments i à i+n-1 et insère les éléments e1, …, ek
.forEach(f) : //Applique la fonction f à tous les éléments
//du tableau qui ne valent pas undefined.
```

Javascript et DOM

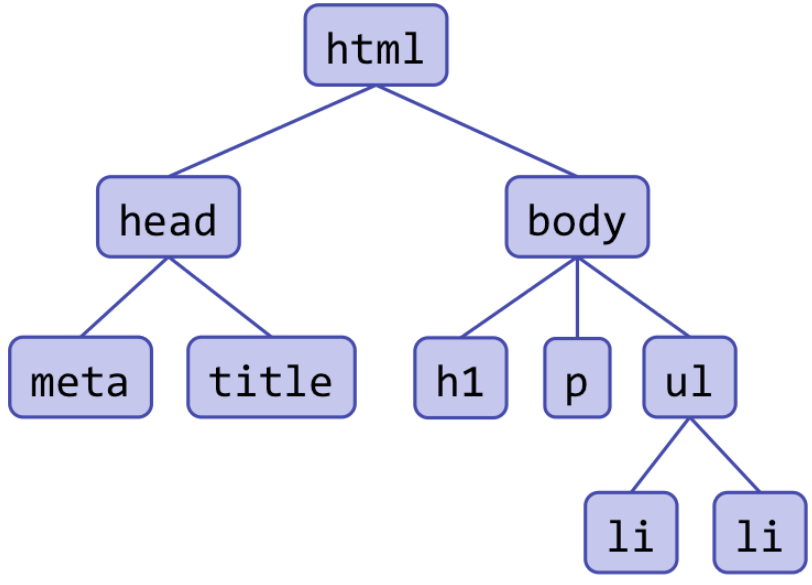
Document Object Model

Le DOM (Document Object Model) est une API orientée objet pour la lecture/manipulation des documents html et xml...

Fournit une représentation structurelle du document et des méthodes pour accéder à son contenu et à sa présentation.

- Ensemble d'interfaces (classes+propriétés).
- Types de base (chaines de caractères,entiers, etc).

La représentation du document est celle d'un arbre.



Interface Node

Represente chaque noeud de l'arbre

```
.ParentNode //Le noeud parent du noeuf courant
.childNodes[n] //Retourne le n-ieme fils du noeud courant
.firstChild //Le premier fils du noeud courant
.lastChild //Le dernier fils du noeud courant
.nextSibling //Permet de récupérer noeud enfant suivant
//d'un noeud.
.prevSibling //Permet de récupérer noeud enfant precedent
//d'un noeud.
.nodeType //Retourne le type du noeud "TR", "IMG",  etc.
.attributes //Retourne les attributs du noeud
```

Plusieurs sous-interfaces de **Node** dont les principales sont :

- Document : interface du noeud racine
- Element : interface des noeuds correspondant à des balises
- Attr : interface des noeuds correspondant à des attributs
- Text : interface des noeuds correspondant à des textes

Javascript et DOM

Attraper le contenu des differentes balises de la page HTML grâce au méthodes de l'interface Document

- `.getElementById("foo")` : récupère l'élement HTML correspondant à la balise d'identifiant "foo"
- `.getElementsByTagName("div")` : récupère sous la forme d'un tableau tout les élements de type div

Récuperer et modifier les attributs de balises HTML

- `.getAttribute(att)` : récupère l'attribut att de l'élement courant
- `.setAttribute(att,nval)` : modifie l'attribut att de l'élement courant en nval

Récuperer et modifier le contenu texte des balise HTML

- `.innerHTML` : récupère le contenu HTML de l'élement courant
- `.appendChild` : ajoute un élement enfant à l'élement courant

Beaucoup d'autres méthodes disséminés dans les interfaces Document, Element, Noeud...etc

Javascript et DOM : exemple

Supposons que la page HTML contienne la balise paragraphe suivante :

```
<p id="intro"> Ceci est l'ancien paragraphe </p>
```

1. On récupère cet élement par son Id

```
var monElement = document.getElementById("Intro");
```

2. On change la propriété de cet élement

```
monElement.style.fontSize="12pt";
```

Ou

```
monElement.innerHTML("Ceci est mon nouveau paragraphe");
```

Javascript et événements

Evenements

- Mécanisme de liaison entre le script et les actions utilisateurs : s'inserent en tant qu'attributs d'un tag html. La valeur de l'attribut est la fonction qui gère l'évenement.

Liste non exhaustive d'évenements :

```
Abort  //l'utilisateur interrompt le chargement de l'image
Change  //l'utilisateur modifie le contenu d'un champ de données.
Click //l'utilisateur clique sur l'élément associé à l'événement
Dblclick //l'utilisateur double clique sur l'élément associé à l'événement
Dragdrop //l'utilisateur effectue un glisser déposer sur la fenêtre
Error //se déclenche lorsqu'une erreur apparaît
Focus //l'utilisateur donne le focus à un élément
Keydown   //l'utilisateur appuie sur une touche de son clavier
keypress  //l'utilisateur maintient une touche de son clavier enfoncée
Keyup //l'utilisateur relâche une touche préalablement enfoncée
Load  //le navigateur de l'utilisateur charge la page en cours
MouseOver //l'utilisateur positionne le curseur de la souris au
//dessus d'un élément
MouseOut //le curseur de la souris quitte un élément
Reset //l'utilisateur efface les données d'un formulaire à l'aide
//du bouton Reset
Select //l'utilisateur sélectionne un texte dans un champ de type "text"
//ou "textarea«
Submit //l'utilisateur clique sur le bouton de soumission d'un formulaire
MouseMove, MouseUp, MouseDown ...
```

Evénements : exemple

```
<html xmlns="http://www.w3c?.org/1999/xhtml" xml:lang="fr">
<head>
      <title> this is my test </title>
      <script type="text/javascript">
      <!
      window.onload = function(){alert("fin chargement page");}
      function testEvent(ev) {
            alert('I was triggered as " type :"+ev.type);
      }
      >
      </script>
</head>
<body onclick="testEvent(event)" onmousemove="testEvent(event)">
      <h1 onclick="testEvent(event)"> Titre </h1>
</body>
</html>
```

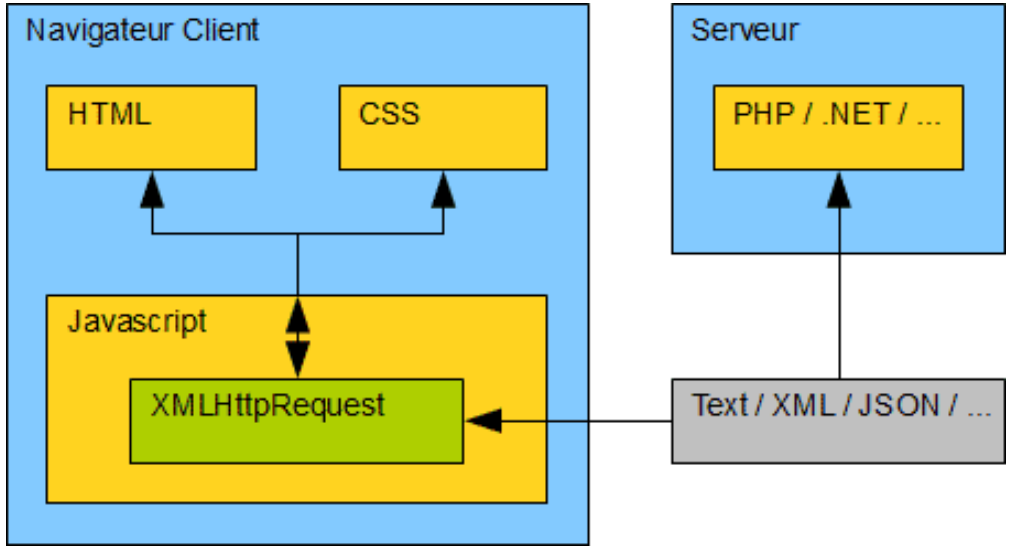
Depuis DOM2, un évenement est associé à un élement du DOM grâce à la méthode addEventListner : `Element.addEventListner('click',function)`

Event est un objet qui contient également de nombreuses proprietés : `Button`, `clientX, clientY` ...

Ajax : Asynchronous JavaScript And XML

Principe général

- Faire communiquer une page Web avec un serveur Web sans occasionner le **rechargement de la page**.
- Echange de donnée avec le server : **Text, HTML, XML, JSON**
- **Asynchrone** : la fonction qui envoie une requête au serveur n'est pas la même que celle qui en recevra la réponse. Processus non bloquant.



XMLHttpRequest

L'objet XMLHttpRequest

- Permet d'envoyer une requête HTTP au serveur, et de récuperer les données renvoyées par celui-ci
- Developé par Microsoft en 1999 sous le nom de XMLHTTP (outlook, IE 5).
- Adopté par les autres navigateurs sous le nom XMLHttpRequest
- Standardisé plus tard par W3C https://www.w3.org/TR/XMLHttpRequest/

Création d'un objet XMLHttpRequest

```
var xhr = new XMLHttpRequest();
```

```
function getXMLHttpRequest() {
   var xhr = null;

   if (window.XMLHttpRequest || window.ActiveXObject) {
        if (window.ActiveXObject) {
           try {
                xhr = new ActiveXObject("Msxml2.XMLHTTP");
           } catch(e) {
                xhr = new ActiveXObject("Microsoft.XMLHTTP");
           }
        } else {
                xhr = new XMLHttpRequest();
               }
   } else {
        alert("Navigateur ne supporte pas XMLHTTPRequest...");
        return null;
   }
        return xhr;
}
```

Envoi d'une requête

```
var xhr = getXMLHttpRequest(); // La fonction getXMLHttpRequest()
//définie dans la partie précédente

xhr.open("GET" || "POST", "cible.php", true);
xhr.send(null);
```

Si la méthode d'envoi est POST, il est necessaire de changer le type MIME avec la commande suivante

```
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

Pour envoyer des variables

```
// Méthode GET
xhr.open("GET", "Cible.php?variable=machin", true);
xhr.send(null);

// Méthode POST
xhr.open("POST", "handlingData.php", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send("variable=machin");
```

État de la requete XMLHttpRequest

L'envoi d'une requête avec l'objet XMLHttpRequest passe par quatre états

- 0 : L'objet a été créé, mais pas encore initialisé (la méthode open n'a pas encore été appelée)
- 1 : L'objet a été créé, mais pas encore envoyé (avec la méthode send)
- 2 : La méthode send vient d'être appelée
- 3 : Le serveur traite les informations et a commencé à renvoyer des données
- 4 : Le serveur a fini son travail, et toutes les données sont réceptionnées

On determine l'etat à l'aide de la propriété `onreadystatechange`

```
var xhr = getXMLHttpRequest();

xhr.onreadystatechange = function() {
    if (xhr.readyState == 4 && (xhr.status == 200 || xhr.status == 0)) {
            alert("OK"); // données réceptionnées
    }
};

xhr.open("GET", "cible.php", true);
xhr.send(null);
```

Récupération des données

Utilisation des deux propriétés suivantes :

- `responseText` : pour récupérer les données sous forme de texte brut
- `responseXML` : pour récupérer les données sous forme d'arbre XML

```
xhr.onreadystatechange = function() {
   if (xhr.readyState == 4 && (xhr.status == 200 || xhr.status == 0))
         alert(xhr.responseText); // Données textuelles récupérées
};
```

Utilisation d'une fonction **callback**

```
function request(callback) {
    var xhr = getXMLHttpRequest();

    xhr.onreadystatechange = function() {
      if (xhr.readyState == 4 && (xhr.status == 200 || xhr.status == 0)) {
            callback(xhr.responseText);
      }
    };
    xhr.open("GET", "cible.php", true);
    xhr.send(null);
}
request(traitement)
```

Exemple

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Techniques AJAX - XMLHttpRequest</title>
<script type="text/javascript" src="oXHR.js"></script>
<script type="text/javascript">

function request(callback) {
    var xhr = getXMLHttpRequest();

    xhr.onreadystatechange = function() {
      if (xhr.readyState == 4 && (xhr.status == 200 || xhr.status == 0)) {
                        callback(xhr.responseText);
      }
    };
    var pseudo = encodeURIComponent(document.getElementById("pseudo").value)
    var nom = encodeURIComponent(document.getElementById("nom").value);

    xhr.open("GET", "XMLHttpRequest_getString.php?Nick=" + pseudo
    + "&Name=" + name, true);
    xhr.send(null);
}
function Liredonnees(Donnees) {
        alert(Donnees);
}
</script>
```

Exemple (suite)

```
</head>
<body>
<form>
  <p>
     <label for="nick">Pseudo :</label>
     <input type="text" id="nick" /><br />
     <label for="name">Prénom :</label>
     <input type="text" id="name" />
  </p>
  <p>
   <input type="button" onclick="request(Liredonnees);" value="Exécuter" />
  </p>
</form>
</body>
</html>
```