Programmation Avancée en Javascript

MOUTAI/BELATTAR/EBOBISSE

s.moutai|m.belattar|y.ebobissedjene@emsi-edu.ma

3éme Année - Ingénierie en Informatique et Réseaux

12 novembre 2024

Table of Contents

JS (Modèle)

```
function (parameters) {
    ... statements ... ;
}
```

- JS permet de définir des fonctions anonymes.
- Une fonction sans nom.
- Elle peut être affectée à une variable, à un handler, etc...

JS (Exemple)

```
window.onload = function() {
let ok = document.getElementById("ok");
ok.onclick = okayClick; };
function okayClick() { alert("booyah"); }
```

ou bien, plus concis, mais peut-être plus difficile à lire :

JS (Exemple)

```
window.onload = function() {
   document.getElementById("ok").onclick = function() {
      alert("booyah");
   };
};
```

JS (Exemple)

```
function everything() {
   let count = 0;
function incr(n) {
   count += n;
}
function reset() {
   count = 0;
}
incr(4);
console.log(count);
}
everything();                    // call the function to run the code
```

Dans l'exemple précédant, tout le code est à l'intérieur d'une fonction.

Les variables et les autres fonctions définies à l'intérieure sont locales à la fonction englobante.

Dans ce code, il n'y a qu'une unique entité globale, c'est la fonction `everything`.

JS (Exemple)

```
window.onload = function() {

   function okayClick() {
      alert("booyah");
   }

   let ok = document.getElementById("ok");
   ok.onclick = okayClick;
};
```

Dans cet exemple, tout le code est à l'intérieur d'une fonction anonyme directement associée à l'évènement `window.onload`.

Dans ce code, il n'y a plus d'entité globale !

Il est possible d'affecter une fonction à une variable et l'appeler à partir de la variable.

JS (Exemple)

```
let test = function(name) {

    if(name==='') {
       alert("La valeur n'a pas été donnée")
      }
   else {
       alert("Bienvenue "+name)
    }
};
# Appel de la fonction
test(prompt('Saisir le nom'));
```

- Les fonctions fléchées proposent une syntaxe plus simple à utiliser
- elles utilisent le symbole =>
- Pas besoin de retourner la variable car les affectations sont faites automatiquement au résultat

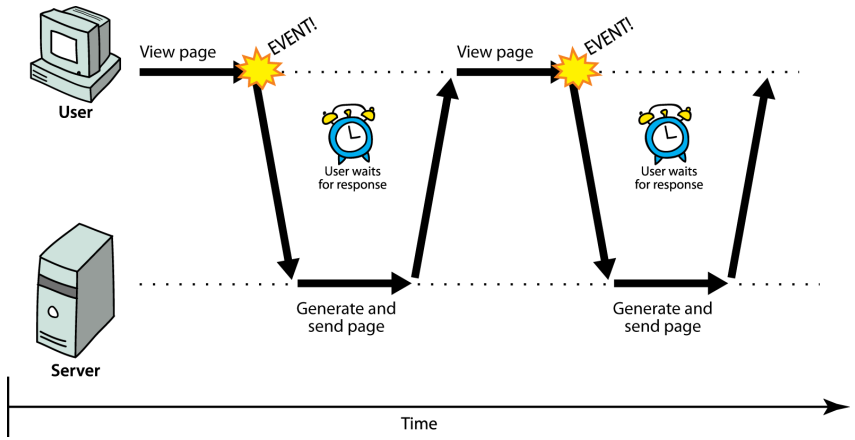
JS (Exemple)

```
let produit = (a, b) => a*b;

# Appel de la fonction
c=produit(4,5);
alert(c)
```

Table of Contents

- 2 instructions sont dites synchrones si la seconde attend la fin de l'exécution de la première.
- La première doit être complétée avant le déclenchement de la seconde.
- Par défaut javascript est conçu pour être synchrone : un processus à la fois



- Synchrone : l'utilisateur doit attendre le chargement de la page.

- Les callbacks ou fonctions de rappel sont des fonctions transmises en paramètres et qui vont être rappelée après un certain temps, ou si certaines conditions sont réunies.
- Peuvent être mises en place à l'aide de **timers**
- **setTimeout(function, delayMS)** : Appelle une fonction après un certain délai.
- **setInterval(function, delayMS)** : Appelle une fonction périodiquement à un intervalle de temps donné.
- **clearTimeout(timerID)**, **clearInterval(timerID)** : Arrête et détruit un timer.

Les fonctions `setTimeout` and `setInterval` retournent une valeur unique qui identifie le *timer*.

- Cette valeur est passée en paramètre à la fonction `clearTimeout` ou `clearInterval` pour arrêter et détruire le *timer*.

Les Timers
Exemple avec setTimeout

HTML (Output)

```
<button id="clickme">Click me!</button>
<span> id="outputText"></span>
```

JS (Exemple)

```
window.onload = function() {
  setTimeout(sayBooyah, 5000);
  alert(" Welcome to this town");
}

function sayBooyah() {                          // called when the timer goes off
  document.getElementById("outputText").innerHTML = "BOOYAH!";
}
```

L'alert sera exécuté avant l'exécution de la fonction sayBooyah

JS (Exemple)

```
let timer = null;                                    // stores ID of interval timer

function delayMsg2() {
   timer = setInterval(mowgli, 1000);
}

function mowgli() {
   document.getElementById("outputText").innerHTML += "Mowgli !"
}
```

JS (Exemple)

```
let timer = null;                                   // stores ID of interval timer

function toggleDelayMessage() {
   if (timer === null) {
      timer = setInterval(mowgli, 1000);
   } else {
      clearInterval(timer);
      timer = null;
   }
}

function mowgli() {
   document.getElementById("outputText").innerHTML += "Mowgli !"
}
```

JS (Exemple)

```
function delayedMultiply() {
   // 6 and 7 are passed to multiply when timer goes off
   setTimeout(multiply, 2000, 6, 7);
}

function multiply(a, b) {
   alert(a * b);
}
```

On place les paramètres après l'intervalle de temps, cela ne marche pas sous IE : il faut créer une fonction spéciale pour ça !

Pourquoi ne pas écrire simplement :

JS (Exemple)

```
setTimeout(multiply(6, 7), 2000);
```

- Fonction de rappel.
- Fonction transmise en argument d'une autre fonction.
- Les callback peuvent être mis en place grâce aux eventListener.

HTML (Output)

```
<button id="clickme">Click me !</button>
<span> id="outputText"></span>
```

JS (Exemple)

```
document.getElementById('clickme').addEventListener('click',callme);
function callme(){
 document.getElementById("outputText").innerHTML ="You called Me";
}
```

- Pour conserver le comportement asynchrone et transmettre des paramètres, il faut utiliser les fonctions fléchées.

HTML (Output)

```
<button id="clickme">Click me !</button>
<span> id="outputText"></span>
```

JS (Exemple)

```
let t= document.getElementById('clickme') ;
t=addEventListener('click',()=>callme('Henry')) ;
function callme(name){
 document.getElementById("outputText").innerHTML =name+"You called
Me" ;
}
```

- L'objet Promise a été introduit pour gérer la programmation asynchrone à partir de 2015.
- il permet de représenter l'état d'une opération asynchrone : **en cours**(pas encore exécutée) , **honorée** (exécutée avec succès), **rompue** (non exécutée)
- les actions adéquates devront donc être prises en compte par la méthode then

JS (Exemple)

```
const prom = new Promise((resolve, reject) => {
setTimeout(()=>alert("Wait"),5000);
});
prom.then(()=>alert('success'), ()=>alert('echec'));
```

1. resolve est appelée si la promesse est tenue
2. reject est appelée dans le cas contraire

- Il est possible de définir les fonctions séparément à l'aide de then (succès) et catch (échec).

JS (Exemple)

```
const prom = new Promise((resolve, reject) => {
setTimeout(()=>alert("Wait"),5000);
});
prom.then(()=>alert('success'));
prom..catch(()=>alert('echec'););
```

- async rend une fonction asynchrone
- cette fonction va alors retourner une promesse
- si cette fonction retourne une valeur, cette valeur sera encapsulée dans la promesse.

JS (Exemple)

```
async function hello(){
return "Hello";
}
hello().then(alert);
```

- await : uniquement dans les fonctions asynchrones définies par async.
- attend le résultat de l'exécution d'une promesse.

JS (Exemple)

```
async function hello(){
const prom = new Promise((resolve, reject) => {
setTimeout(()=>alert("Wait"),5000);
});
return await prom;
}
function resolveAfter2Seconds(x) {
return new Promise((resolve) => { setTimeout(() =>{
resolve(x); }, 2000);
});
}
```

- await : uniquement dans les fonctions asynchrones définies par async.
- attend le résultat de l'exécution d'une promesse.

JS (Exemple)

```
async function f1() {
var x = await resolveAfter2Seconds(10);
console.log(x); // 10 }
f1();
```

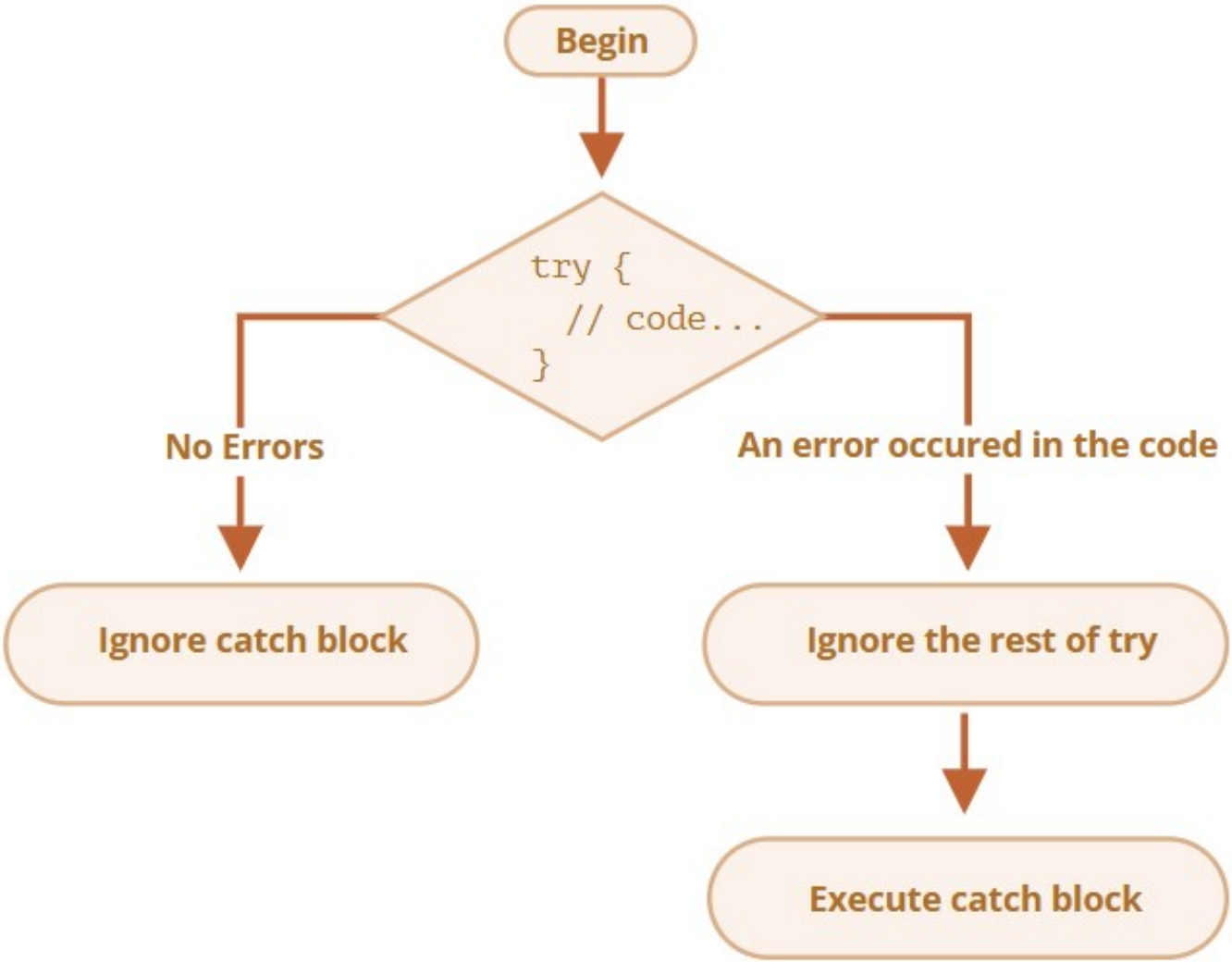
Gestion des erreurs
try-catch

- Lors de l'exécution d'un script, des erreurs peuvent arriver (fichier non chargé, image manquante, mauvaise donnée saisie, etc...)
- Objetcif : capturer les erreurs et les gérer

JS (Syntaxe)

```
try {
    // code...
} catch (err) {
    // Gestion des erreurs }
```

Gestion des erreurs
try-catch



Le bloc finally s'exécute qu'il y ait erreur ou pas.

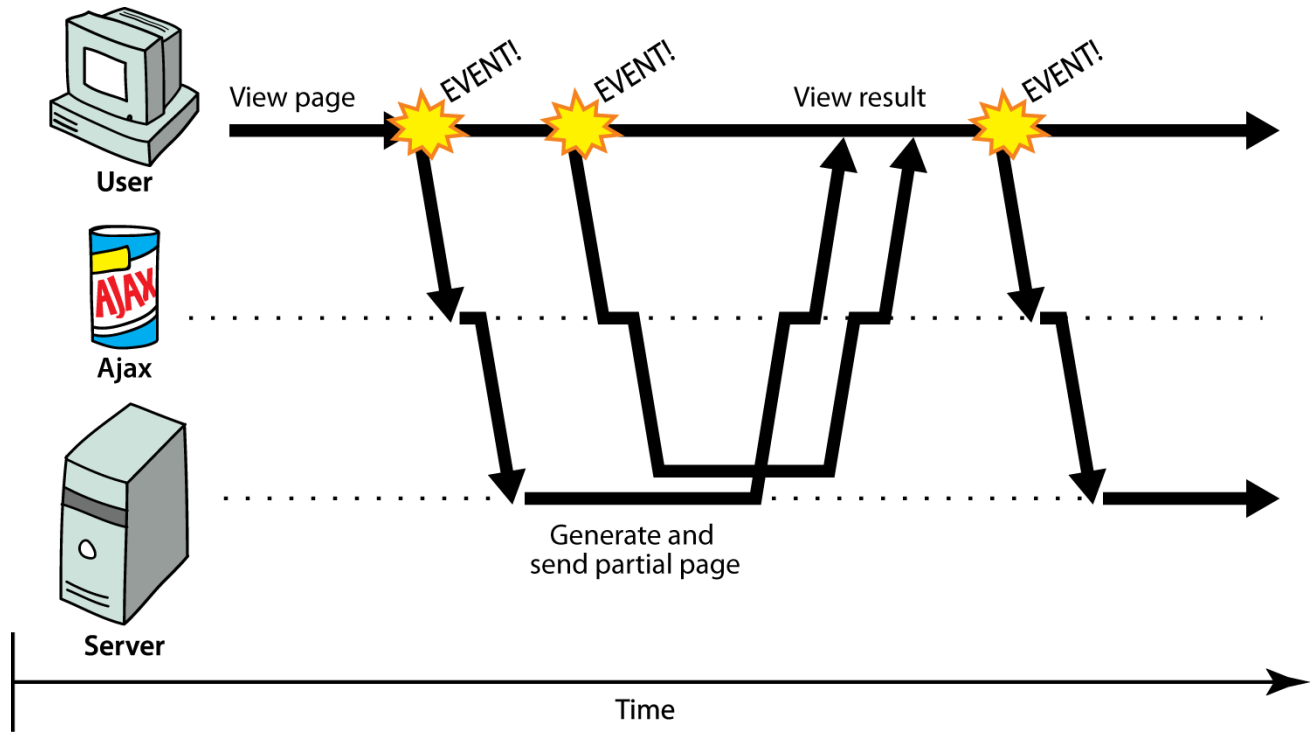
JS (Exemple)

```
a=prompt("Entrer un entier a");
b=prompt("Entrer un entier b");
try{
    c=a/b;
    k;
} catch (err) {
    console.log(err.name);   # affiche le nom de l'erreur
    console.log(err.message);   # affiche le message lié à l'erreur
}
finally {
    console.log("On continue malgré tout");
}
```

Table of Contents

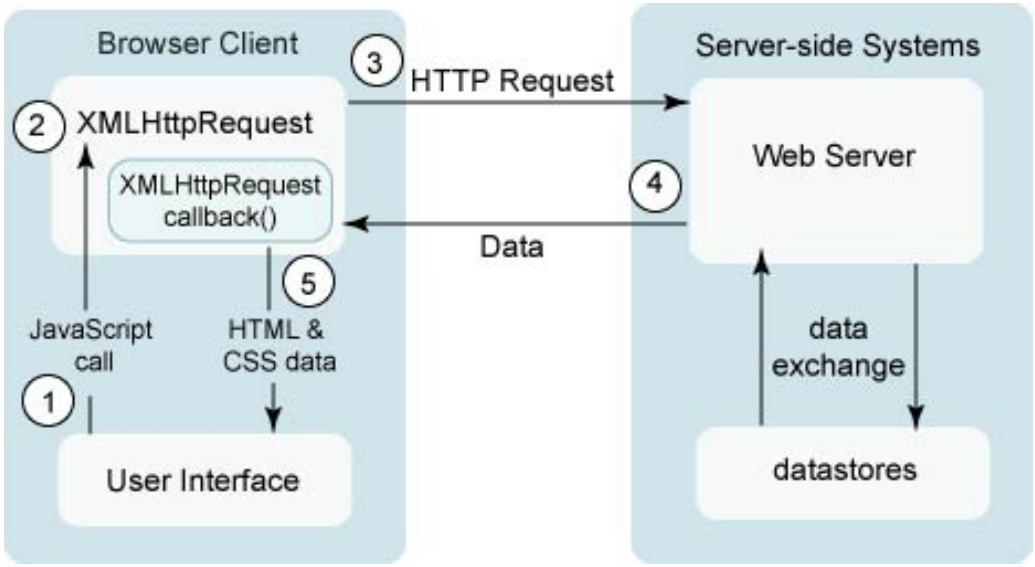
- Application web : un site web dynamique qui imite le comportement d'une application de bureau.
  - Offre une interaction continue à l'utilisateur plutôt qu'une suite de pages différentes.
  - Exemples : Gmail, Google Maps, Google Docs and Spreadsheets, Flickr, A9.

- Ajax : Asynchronous JavaScript and XML
  - Pas un langage de programmation mais une façon d'utiliser JavaScript.
  - Permet de récupérer des données du serveur en tâche de fond.
  - Permet de mettre à jour une page dynamiquement sans la recharger.
  - Un modèle différent du modèle "cliquer-attendre-rafraîchir".



- Asynchrone : l'utilisateur peut interagir avec la page pendant le chargement de nouvelles données.
  - Un autre modèle pour les pages web.

- JavaScript contient l'objet `XMLHttpRequest` qui peut envoyer des requêtes au serveur.
  - Disponible dans tous les navigateurs.
  - Sauf dans les vieilles versions (en particulier IE6-).
  - Pourtant inventé par Microsoft pour les liens entre IE et OutLook...
- Les requêtes sont asynchrones (exécutées en tâche de fond).
- Les données récupérées peuvent être utilisées pour modifier le HTML via JavaScript/DOM.



1. L'utilisateur clique et cet évènement appelle une fonction.
2. Cette fonction fabrique un objet XMLHttpRequest.
3. L'objet XMLHttpRequest invoque un script sur le serveur.
4. Le script produit de la donnée et la transmet à l'obet XMLHttpRequest.
5. XMLHttpRequest déclenche un évènement à la réception des données.
   - La fonction associée à cet évènement est appelée.
   - On appelle cette fonction un callback.
6. Le callback traîte les données reçues et modifie le HTML en conséquence.

- Ajax ne fonctionne qu'au travers d'un serveur web.
  - *(ne fonctionne pas en mode* `file` *)*.
- Ajax ne peut effectuer des requêtes que vers le même serveur qui a servi la page dans laquelle il se trouve.
  - `http ://www.foo.com/a/b/c.html` ne peut faire des requêtes Ajax qu'à l'adresse `http ://www.foo.com` .

Un constructeur écrit pour simplifier les requêtes Ajax.

HTML (Exemple)

```
new simpleAjax(url, method, params, onSuccess, onFailure)
```

- `url` : le script invoqué côté serveur (la requête).
- `method` : la methode GET ou POST (`"get"` ou `"post"`).
- `params` : les paramètres de la requête (une chaîne de caractères de la forme `param1=val1&param2=val2&...`
- `onSuccess` : la fonction appelée au retour de la requête (et qui va traîter la valeur retournée).
- `onFailure` : la fonction appelée en cas d'erreur.

Les exemples suivants sont founis pour comprendre comment utiliser l'objet simpleAjax.

1. Exemple 1 : Comment récupérer périodiquement des informations (aléatoires) depuis le serveur (requêtes Ajax sans paramètre).
2. Exemple 2 : Comment échanger des informations simples avec le serveur (requêtes Ajax avec paramètres, sans JSON).
3. Exemple 3 : Comment échanger des informations structurées avec le serveur (requêtes Ajax avec paramètres et en JSON).

Pour tester ces exemples, téléchargez l'archive **exemples-ajax.zip** et dézippez-la dans votre répertoire web.

Table of Contents

JavaScript Object Notation (JSON) est un format de données pour encoder des valeurs JavaScript.

- Inventé par le gourou JavaScript **Douglas Crockford** (Yahoo).
- Intégré dans tous les navigateurs et disponible sous forme de bibliothèque dans tous les langages de programmation.
- Très populaire et très utilisé à cause de sa simplicité.

Les fonctions `setTimeout` and `setInterval` retournent une valeur unique qui identifie le *timer*.

- Cette valeur est passée en paramètre à la fonction `clearTimeout` ou `clearInterval` pour arrêter et détruire le *timer*.

JS (Modèle)

```
let name = {
fieldName : value,
...
};
```

JS (Exemple)

```
let pt = {
x : 4,
y : 3 };
pt.z = -1;
alert("(" + pt.x + ", " + pt.y + ", " + pt.z + ")");      // (4, 3, -1)
```

- En JavaScript, on peut créer des objets sans créer une classe (juste en utilisant les caractères '{' et '}').
- On peut leur ajouter dynamiquement des attributs (comme `z`).

JS (Exemple)

```
var person = {
   name : "Philip J. Fry",                          // string
   age : 23,                                        // number
   "weight" : 172.5,                                // number
   friends : ["Farnsworth", "Hermes", "Zoidberg"],  // array
getBeloved : function() { return this.name + " loves Leela"; }
};
alert(person.age);                                  // 23
alert(person["weight"]);                            // 172.5
alert(person.friends[2]));                          // Zoidberg
alert(person.getBeloved());                         // Philip J. Fry loves Leela
```

- Un objet peut avoir des méthodes qui utilise `this` comme référence à l'objet sur lequel on les appelle.
- On peut accéder aux attributs par `.fieldName` ou `["fieldName"]`.

JSON (Exemple)

```
{
   "private" : "true",
   "from" : "Chakib Hamid (chakib@hamid.com)",
   "to" : [
            "Anis Rochdi (anis@rochdi.com)",
            "Aya Sophia (aya@sophia.com)"
          ],
   "subject" : "Tomorrow's \" Birthday Bash \" event !",
   "message" : {
            "language" : "english",
            "text" : "Hey guys, don't forget to call me this weekend !"
   }
}
```

JS (Exemple)

```
var student = {                                    // pas d'affectation
   "first_name" : 'Bart',                          // chaînes entre " (et non ')
   last_name : "Simpson",                          // attributs entre "
   "birthdate" : new Date("April 1, 1983"),        // pas d'instance de classe
   "enroll" : function() {                         // pas de fonction
      this.enrolled = true;
   }
};
```

- Les chaînes doivent être délimitées avec le caractères `"`.
- Les attributs (les champs) doivent être placés entre des `"`.
- Types interdits : `Function`, `Date`, `RegExp`, `Error`.
- Types autorisés : `Number`, `String`, `Boolean`, `Array`, `Object`, `Null`.
- Il existe des validateurs pour JSON : **JSONLint**, **JSON Formatter & Validator**, **Free Formatter**, **JSON Validator**.

- `JSON.parse(string)` : Convertit une chaîne JSON en valeur JavaScript et retourne cette valeur.
- `JSON.stringify(object)` : Convertit une valeur JavaScript en chaîne JSON et retourne cette chaïne.

Pour envoyer un paramètre complexe au serveur :

1. On fabrique un objet JavaScript qui encode le paramètre.
2. A l'aide la fonction `JSON.stringify` on convertit l'objet JavaScript en chaîne JSON.
3. On effectue un requête Ajax au serveur avec la chaîne JSON en paramètre.

Pour recevoir la réponse complexe du serveur :

1. On reçoit la donnée en retour au format JSON.
2. A l'aide la fonction `JSON.parse` on convertit la chaîne JSON en objet JavaScript.
3. On utilise cette valeur pour mettre à jour le HTML.

- **json_decode(string)** : Convertit une chaîne JSON en valeur PHP sous la forme d'un tabmeau associatif et retourne cette valeur (comme `JSON.parse` en JavaScript).
- **json_encode(object)** : Convertit une valeur PHP (valeur primitive, tableau associatf ou objet) en chaîne JSON et retourne cette chaïne (comme `JSON.stringify` en JavaScript).
- **NB : json_encode** convertit les tableaux associatifs en objets et les tableaux indicés en tableaux.

PHP (Exemple)

```php
<?php
  $data = array(
    "library" => "Odegaard",
    "category" => "fantasy",
    "year" => 2012,
    "books" => array(
      array("title" => "Harry Potter",
        "author" => "J.K. Rowling"),
      array("title" => "The Hobbit",
        "author" => "J.R.R. Tolkien"),
      array("title" => "Game of Thrones",
        "author" => "George R. R. Martin"),
    )
  );
  header("Content-type : application/json");
  echo <em>json_encode($data)</em>;
?>
```

Le script précédent produit la valeur JSON suivante :

JSON (Exemple)

```
{
   "library" : "Odegaard",
   "category" : "fantasy",
   "year" : 2012,
   "books" : [
      "title" : "Harry Potter", "author" : "J.K. Rowling",
      "title" : "The Hobbit", "author" : "J.R.R. Tolkien",
      "title" : "Game of Thrones", "author" : "George R. R. Martin",
      "title" : "Dragons of Krynn", "author" : "Margaret Weis"
               ]
}
```