

Introduction à JavaScript



Slides de Sébastien Gagné
Programmation Web - Client Riche



1

Plan du cours

I. Généralités sur JavaScript

1. Bref historique de JS
2. Environnement de travail
3. Caractéristiques générales de JS

II. Variables en JavaScript

1. Les types principaux
2. Déclaration des variables
3. Portée des variables
4. Opérateurs classiques
5. Quelques méthodes intéressantes

III. Tableaux en JavaScript

1. Déclarations possibles d'un tableau
2. Contenu d'un tableau et accès au contenu
3. Parcours d'un tableau
4. Méthodes et attributs classiques d'un tableau
5. Autres méthodes d'un tableau
6. Tableaux associatifs ?

IV. Fonctions en JavaScript

1. Fonctions natives importantes
2. Fonctions construites
3. Fonctions anonymes

V. Objets en JavaScript

Programmation Web - Client Riche



2

Généralités sur JavaScript

1. Bref historique

a) Années 1990 - Dynamic HTML – effets sur les pages web

- Langage écrit en 1995 par Brendan EICH chez Netscape, pour associer des scripts à des éléments HTML.
- Permet d'obtenir des pages web dynamiques, en interaction avec l'utilisateur. Action sur la structure du document HTML.
- Adoption générale et très rapide du concept de dynamisation du HTML par des scripts côté client.
- Standard 96-97 (ECMAScript) aujourd'hui version 8
- Déclinaison d'ECMAScript suivant les navigateurs :
 - Mozilla : JavaScript
 - Microsoft : JScript



Programmation Web - Client Riche



3

Généralités sur JavaScript

1. Bref historique

b) Années 2000 – Bibliothèques évoluées

- JQuery, MooTools, AngularJS, ... : proposent un ensemble de fonctions, ou même un cadre de travail complet pour JavaScript.
- AJAX (utilisation « asynchrone » de JavaScript pour gérer des appels au serveur de données). Voir TD5 et suivants.

c) Années 2010 – Ère moderne

- Évolution de JavaScript : utilisation du langage côté serveur (retour aux origines).
- Tendance actuelle : un seul langage dans la pile web, par exemple remplacer PHP par JavaScript.
- Node.js pour des serveurs web écrits en JavaScript.

Programmation Web - Client Riche



4

Généralités sur JavaScript

2. Environnement de travail

a) la console du navigateur

Endroit idéal pour tester le code, en interaction directe avec la page web. Outil indispensable. Les exemples du cours sont testés dans la console.

b) l'éditeur de texte

Comme pour tous les autres langages web, on peut se contenter d'un éditeur de texte pour coder.

c) sites dédiés

Certains sites permettent l'élaboration et le test du code client : html, css et JavaScript. Par ex :

- <https://codepen.io/>
- <https://jsbin.com/>

Généralités sur JavaScript

3. Caractéristiques générales de JavaScript

a) langage qui dynamise les pages web côté client

Au moyen de scripts interprétés au niveau du navigateur, la page web est rendue dynamique côté client (voir TD1), souvent par une gestion des événements (clics, ...). Ceci est un point de vue différent de la dynamique côté serveur (PHP), où on gère des informations envoyées (formulaires) ou en provenance de la base de données.

b) langage interprété

JavaScript est interprété au niveau du navigateur, sans la moindre compilation. L'exécution des scripts dépend de l'activation, côté client, de l'interpréteur JavaScript.

Variables en JavaScript

1. Les types principaux

- JavaScript propose 7 types différents, nous en utiliserons essentiellement 4 :
 - Number (les nombres, quels qu'ils soient)
 - String (chaînes de caractères)
 - Boolean (les booléens)
 - Object (tous les objets JavaScript)
- Les 3 autres types (Null, Undefined et Symbol) sont pour nous moins communs.
- Le typage JavaScript est :
 - Faible : Pas de type indiqué à la déclaration.
 - Dynamique : Le type d'une variable peut changer.

Variables en JavaScript

2. Déclaration des variables

- Les variables se déclarent par les mots-clés `var`, `let` ou `const`.
- La tendance actuelle est l'utilisation du mot-clé `let`. Nous privilégierons ce mot-clé.

```
> let bianca = "Bianca Castafiore";
```

- Le mot-clé `const` fonctionne comme `let`, à ceci près que la valeur ne peut pas être réaffectée après initialisation.

```
> const pi = 3.141592653589793
< undefined
> pi = 3
```

```
✖ ▶ Uncaught TypeError: Assignment to constant variable.
   at <anonymous>:1:4
```

Variables en JavaScript

3. Portée des variables

a) variables locales

Une variable déclarée avec `let` a pour portée le bloc contenant (fonction, boucle `for`, ...).

```
> for(let i = 1; i < 4; i++) {  
  console.log(i);  
}  
1  
2  
3  
< undefined  
> console.log(i);  
✖ Uncaught ReferenceError: i is not defined  
  at <anonymous>:1:13
```

```
> let i = 0  
< undefined  
> for(i = 0; i < 4; i++) {  
  console.log(i);  
}  
0  
1  
2  
3  
< undefined  
> console.log(i)  
4
```

Variables en JavaScript

3. Portée des variables

a) variables locales

Une variable déclarée avec `let` a pour portée le bloc contenant (fonction, boucle `for`, ...).

```
> let i = 1  
< undefined  
> function f() {  
  let j = 2;  
  console.log(j);  
}  
< undefined  
> f()  
2  
< undefined  
> console.log(i)  
1  
< undefined  
> console.log(j)  
✖ Uncaught ReferenceError: j is not defined  
  at <anonymous>:1:13
```

Variables en JavaScript

3. Portée des variables

b) variables globales

Une variable déclarée dans la partie principale du script a donc pour portée tout le script : c'est une variable globale

```
> let i_global = 0;  
< undefined  
> function f(nb) {  
  i_global = nb;  
}  
< undefined  
> f(8)  
< undefined  
> i_global  
< 8
```

Variables en JavaScript

4. Opérateurs classiques

a) opérateurs arithmétiques (entre variables de type Number)

Opérateur	Opération
+	addition
-	soustraction
*	multiplication
/	division
%	modulo
**	puissance

On peut aussi utiliser les opérateurs classiques `+=`, `-=`, `*=`, `/=`, `%=` et `**=`

Variables en JavaScript

4. Opérateurs classiques

b) opérateurs logiques (entre variables de type Boolean)

Opérateur	traduction
ET	&&
OU	
NON	!
égalité en valeur	=
égalité en valeur et en type	===
différence en valeur	!=
différence en valeur ou en type	!==

Variables en JavaScript

4. Opérateurs classiques

c) concaténation (entres variables de type String)

La concaténation de chaînes de caractères se fait au moyen de l'opérateur + mais...

ATTENTION : JavaScript est permissif...
Pour maîtriser le résultat de la concaténation, faire attention à l'interprétation de l'opérateur +

opération	résultat
"abc" + "de"	"abcde"
"3" + 1	"31"
3 + 1 + "5"	"45"
"5" + 3 + 1	"531"

Variables en JavaScript

5. Quelques méthodes intéressantes

a) type Number (voir le __proto__ en détail !)

```
> let x = 3.14159265358979;
< undefined
> typeof(x)
< "number"
> x.__proto__
< Number {0, constructor: f, toExponential: f, toFixed: f, toPrecision: f, ...} ⓘ
  ▶ constructor: f Number()
  ▶ toExponential: f toExponential()
  ▶ toFixed: f toFixed()
  ▶ toPrecision: f toPrecision()
  ▶ toString: f toString()
  ▶ valueOf: f valueOf()
> x.toPrecision(4)
< "3.142"
> x.toFixed(4)
< "3.1416"
> x.toString()
< "3.14159265358979"
```

Variables en JavaScript

5. Quelques méthodes intéressantes

b) type String (voir VRAIMENT le __proto__ en détail !)

```
> let password = "Abc12$DEf$a!";
< undefined
> password.__proto__
< String {"", constructor: f, anchor: f, big: f, blink: f, ...}
> password.length
< 12
> password.toUpperCase()
< "ABC12$DEF$a!"
> password.toLowerCase()
< "abc12$def$a!"
> password.charAt(6)
< "D"
> password.indexOf("$")
< 5
> password.replace("A","Z")
< "Zbc12$DEf$a!"
> password.split("E")
< (2) ["Abc12$D", "f$a!"]
```

Variables en JavaScript

5. Quelques méthodes intéressantes

b) type String

On peut aussi faire du remplacement de variables dans des chaînes de caractères.
Il faut utiliser le délimiteur ` (accent grave).

```
> let nom = 'Juste Leblanc';  
let p = `

Bonjour ${nom}  
</p>`;  
console.log(p);  
  
<p>  
  Bonjour Juste Leblanc  
</p>


```

Tableaux en JavaScript

1. Déclarations possibles d'un tableau

Les tableaux JavaScript peuvent être déclarés par un appel à un constructeur de l'objet natif de JavaScript Array :

```
let tab = new Array("bonjour","salut","hello");
```

Mais il est plus simple de les déclarer par :

```
let tab = ["bonjour","salut","hello"];
```

Tableaux en JavaScript

2. Contenu d'un tableau et accès au contenu

- Même si concrètement les tableaux utilisés sont plutôt «monotypes», on peut envisager des tableaux JavaScript contenant des éléments de types variés

```
let tab = ["bonjour",3.14159,true,[1,2,"salut"],6];
```

- On accède à un élément de manière classique, par un système d'indices à partir de 0.

tab[0]	→	"bonjour"
tab[2]	→	true
tab[3]	→	Array(1,2,"salut")
tab[3][1]	→	2
tab[3][2]	→	"salut"

Tableaux en JavaScript

3. Parcours d'un tableau

- On peut parcourir un tableau par une boucle for classique qui utilise la longueur du tableau :

```
> let tab = ["bonjour","hello","salut","coucou"]  
< undefined  
> for(let i = 0; i < tab.length; i++) {  
  console.log("mot n°" + i + " ... " + tab[i]);  
}  
  
mot n°0 ... bonjour  
mot n°1 ... hello  
mot n°2 ... salut  
mot n°3 ... coucou
```

Tableaux en JavaScript

3. Parcours d'un tableau

- On peut aussi parcourir un tableau par une boucle for particulière :

```
> let tab = ["bonjour", "hello", "salut", "coucou"]
< undefined
> for(let mot of tab) {
  console.log("mot courant ... " + mot);
}
```

mot courant ...	bonjour
mot courant ...	hello
mot courant ...	salut
mot courant ...	coucou

Tableaux en JavaScript

4. Méthodes et attributs classiques d'un tableau

a) insertions

- en fin de tableau : `tab.push(elt1, elt2, ...)`
- en début de tableau : `tab.unshift(elt1, elt2, ...)`

Ces 2 méthodes retournent la nouvelle taille de tab

- en général : `tab.splice(i, j, elt1, elt2, ...)`
 - i : endroit d'insertion
 - j : nombre d'éléments à supprimer à partir de i
 - elt1, elt2, ... : éléments à insérer à partir de i

Cette méthode retourne le sous-tableau composé des j éléments supprimés

Tableaux en JavaScript

4. Méthodes et attributs classiques d'un tableau

b) suppressions

- en fin de tableau : `tab.pop()`
- en début de tableau : `tab.shift()`

Ces 2 méthodes retournent l'élément supprimé

c) extractions

- en fin de tableau : `tab.slice(i)`
retourne le sous-tableau des indices k, $k \geq i$
- en milieu de tableau : `tab.slice(i, j)`
retourne le sous-tableau des indices k, $i \leq k < j$

Tableaux en JavaScript

4. Méthodes et attributs classiques d'un tableau

d) agglomération

`tab1.concat(tab2)`

retourne le tableau des éléments de tab1 puis tab2

e) concaténation

`tab.join("/")`

retourne la chaîne de caractères obtenue par concaténation des éléments de tab, séparés par le caractère passé en argument.

f) longueur

longueur du tableau : `tab.length`

Tableaux en JavaScript

5. Autres méthodes d'un tableau

Le `__proto__` d'un tableau est riche et sa lecture vous montrera d'autres méthodes intéressantes...

```
> let tab = ["coucou", "hello", "bonjour", "salut"]
< undefined
> tab.__proto__
< [constructor: f, concat: f, copyWithin: f, fill: f, find: f, ...]
  length: 0
  ▶ constructor: f Array()
  ▶ concat: f concat()
  ▶ copyWithin: f copyWithin()
  ▶ fill: f fill()
  ▶ find: f find()
  ▶ findIndex: f findIndex()
  ▶ lastIndexOf: f lastIndexOf()
  ▶ pop: f pop()
  ▶ push: f push()
  ▶ reverse: f reverse()
  ▶ shift: f shift()
  ▶ unshift: f unshift()
  ▶ slice: f slice()
  ▶ sort: f sort()
```

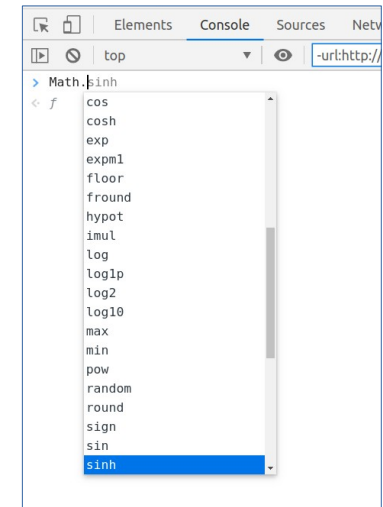
Fonctions en JavaScript

1. Fonctions natives importantes

JavaScript propose un ensemble de fonctions natives qu'on utilisera régulièrement (voir TD1)

Quelques exemples :

- `Math.random()`
- `Math.floor(3.14)`
- `"Dupont".replace("t", "d")`



Fonctions en JavaScript

2. Fonctions construites

On peut aussi construire nos propres fonctions, comme dans le TD1.

Utilisées dans la gestion des événements (clics de souris,...), elles se déclencheront si l'événement en question se produit. C'est ainsi qu'on rendra dynamiques (côté client) nos pages web.

La syntaxe habituelle :

```
> function suivant(n) {
  // Nombre suivant modulo 6
  return (n+1) % 6;
}
```

```
> suivant(1)
< 2
> suivant(4)
< 5
> suivant(6)
< 1
```

Fonctions en JavaScript

3. Utilisation classique d'une fonction

Le cas classique (pas le plus évolué) d'utilisation d'une fonction est décrit dans le code suivant :

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <meta charset="utf-8">
5   <title>utilisation de fonction</title>
6   <link rel="stylesheet" type="text/css" href="style.css">
7 </head>
8 <body>
9   <p id="p1" onclick="f1();">paragraphe 1</p>
10  <p id="p2" ondblclick="f2();">paragraphe 2</p>
11  <script type="text/javascript">
12    function f1() {
13      alert("vous avez cliqué sur p1");
14    }
15    function f2() {
16      alert("vous avez double-cliqué sur p2");
17    }
18  </script>
19 </body>
20 </html>
```

Fonctions en JavaScript

4. Fonctions anonymes

On peut déclarer une fonction sans lui donner de nom explicite, elle est donc anonyme.

Cette situation peut se produire dans 3 cas :

- On déclare la fonction anonymement, mais on l'affecte à une variable... Ce qui revient en gros à donner un nom à la fonction...

```
> let toto = function() {  
  console.log("coucou");  
}  
  
> toto()  
coucou  
  
> toto  
< f () {  
  console.log("coucou");  
}
```

Fonctions en JavaScript

4. Fonctions anonymes

On peut déclarer une fonction sans lui donner de nom explicite, elle est donc anonyme.

Cette situation peut se produire dans 3 cas :

- On «auto-invoque» la fonction, c'est-à-dire on l'exécute immédiatement après sa déclaration.
- La fonction est alors entourée de parenthèses, et suivie d'une paire de parenthèses, ce qui donne une syntaxe assez inhabituelle :

```
> (function() {console.log("coucou");})();  
coucou
```

Fonctions en JavaScript

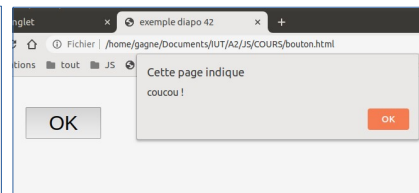
4. Fonctions anonymes

On peut déclarer une fonction sans lui donner de nom explicite, elle est donc anonyme.

Cette situation peut se produire dans 3 cas :

- On utilise la fonction anonyme associée à un événement (LE CAS CLASSIQUE), par ex un clic :

```
1 <!DOCTYPE html>  
2 <html>  
3 <head>  
4 </head>  
5 <body>  
6 <input type="button" id="bouton" value="OK">  
7 <script type="text/javascript">  
8   let b = document.getElementById("bouton");  
9   b.addEventListener("click",function() {  
10     alert("coucou !");  
11   })  
12 </script>  
13 </body>  
14 </html>
```



Objets en JavaScript

- JavaScript est orienté objet, nous allons donc nous retrouver en terrain relativement connu. Mais il y a quelques différences avec Java par exemple...
- En Java, on définit de façon statique les classes qui servent à instancier les objets. Les objets sont instanciés au moyen des classes par appel à un constructeur :

```
Personne p = new Personne("toto")
```


Objets en JavaScript

- Les objets JavaScript ont la particularité de ne pas dépendre d'une définition de classe comme en Java.
- Les versions récentes de JavaScript adoptent néanmoins la maquette de classe mais les objets JavaScript gardent une vraie nature indépendante.
- Ils n'instancient pas de façon pure et dure une maquette de classe, mais sont insérés dans une chaîne de prototypage, qui permet de savoir quel est leur héritage et la généalogie de cet héritage.
- Il faut retenir qu'un objet JavaScript n'est pas contraint par un modèle de classe, mais nos habitudes de prog objet de Java nous permettront de reproduire certaines pratiques classiques...

Objets en JavaScript

a) création d'un objet de façon littérale

- On peut définir un objet en donnant des paires clés-valeurs :

```
> let p = {nom : "Haddock", prenom : "Archibald"}
< undefined
> p.nom
< "Haddock"
> p.prenom
< "Archibald"
```

p a été défini de façon littérale et complètement indépendante.
Il peut être complété à tout moment :

```
> p.profession = "marin"
< "marin"
> p
< {nom: "Haddock", prenom: "Archibald", profession: "marin"}
```

Objets en JavaScript

a) création d'un objet de façon littérale

On peut aussi créer l'objet avec des méthodes ou encore les ajouter après coup :

```
> let p = {nom : "Haddock", prenom : "Archibald"}
> p.profession = "marin"
> p.parler = function() {
  console.log("mille sabords !");
}
> p
< {nom: "Haddock", prenom: "Archibald", profession: "marin", parler: f}
> p.parler()
mille sabords !
```

Objets en JavaScript

b) création d'un objet par une fonction constructeur

Cette méthode rappelle ce qu'on utilise en Java, mais la déclaration préalable des attributs n'a pas de sens puisqu'on peut en ajouter à tout moment...

```
> function Personne(nom, prenom, profession, phrase) {
  this.nom = nom;
  this.prenom = prenom;
  this.profession = profession;
  this.parler = function() {
    console.log(phrase);
  }
}
< undefined
> let capitaine = new Personne("haddock", "Archibald", "marin", "mille sabords !");
< undefined
> capitaine.nom
< "haddock"
> capitaine.parler()
mille sabords !
< undefined
```

Objets en JavaScript

b) création d'un objet par une fonction constructeur

On peut de nouveau enrichir l'objet :

```
> capitaine.vices = ["tabac", "alcool", "mauvaise humeur"];
< ▶ (3) ["tabac", "alcool", "mauvaise humeur"]
> capitaine
< ▼ Personne {nom: "haddock", prenom: "Archibald", profession: "marin", vices: Array(3), parler: f}
  nom: "haddock"
  prenom: "Archibald"
  profession: "marin"
  ▶ parler: f ()
  ▼ vices: Array(3)
    0: "tabac"
    1: "alcool"
    2: "mauvaise humeur"
    length: 3
    ▶ __proto__: Array(0)
  ▶ __proto__: Object
> capitaine.vices[1]
< "alcool"
```

Objets en JavaScript

c) création d'un objet selon un modèle de classe

Façon plus classique de coder :

```
> class Personne {
  // constructeur
  constructor(nom, prenom, profession) {
    this.nom = nom;
    this.prenom;
    this.profession = profession;
  }
  // autres méthodes
  parler (phrase) {
    console.log(phrase);
  }
}

let capitaine = new Personne("Haddock", "Archibald", "marin");
capitaine.parler("mille sabords !");
mille sabords !
```

Objets en JavaScript

c) création d'un objet selon un modèle de classe

```
> class Personne {
  constructor(nom, prenom, profession) {
    this.nom = nom;
    this.prenom = prenom;
    this.profession = profession;
  }
  parler = function(phrase) {
    console.log(phrase);
  }
}
< undefined
> let capitaine = new Personne("Haddock", "Archibald", "marin")
< undefined
> capitaine.vices = ["tabac", "alcool", "mauvaise humeur"]
< ▶ (3) ["tabac", "alcool", "mauvaise humeur"]
> let professeur = new Personne("Tournesol", "Tryphon", "savant fou")
< undefined
> professeur
< ▶ Personne {nom: "Tournesol", prenom: "Tryphon", profession: "savant fou", parler: f}
```

Objets en JavaScript

d) parcours des attributs et méthodes d'un objet

On peut parcourir l'objet par une boucle for :

```
> let gaston = {
  nom: "Lagaffe",
  prenom: "Gaston",
  profession: "gaffeur"
}
< undefined
> for(att in gaston) {
  console.log("Gaston possède l'attribut " + att);
}
Gaston possède l'attribut nom
Gaston possède l'attribut prenom
Gaston possède l'attribut profession
```

Objets en JavaScript

e) Tableaux associatifs ?

Les objets JavaScript peuvent être vus comme des «tableaux associatifs» en lisant autrement leurs attributs :

```
let p = {nom:"Dupont",prenom:"Pierre",age:35}
```

```
> p.nom      →      "Dupont"
> p.prenom   →      "Pierre"
> p.age      →      35

> p["nom"]   →      "Dupont"
> p["prenom"] →      "Pierre"
> p["age"]   →      35
```

Objets en JavaScript

f) langage basé sur les prototypes

- JavaScript n'élabore pas les objets sur le concept de classes statiques, mais sur le concept plus complexe de prototype.
- capitaine et professeur sont construits selon le prototype de `Personne` (et donc héritent des attributs et méthodes).

- On peut modifier le prototype après coup :

```
> Personne.prototype.aurevoir = function(phrase) {
    alert(phrase);
}
```

- capitaine et professeur héritent **dynamiquement** des méthodes du prototype de `Personne`...

Objets en JavaScript

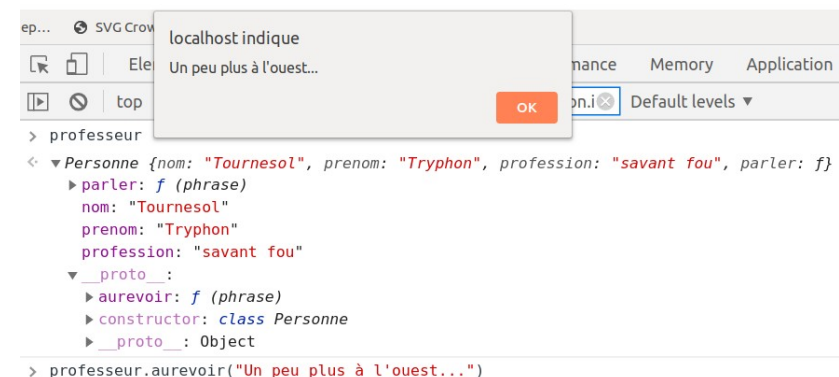
f) langage basé sur les prototypes



```
> capitaine
< ▼ Personne {nom: "Haddock", prenom: "Archibald", profession: "marin", vices: Array(3), parler: f}
  ▶ parler: f (phrase)
    nom: "Haddock"
    prenom: "Archibald"
    profession: "marin"
  ▶ vices: (3) ["tabac", "alcool", "mauvaise humeur"]
  ▼ __proto__:
    ▶ aurevoir: f (phrase)
    ▶ constructor: class Personne
    ▶ __proto__: Object
> capitaine.aurevoir("Bon vent, moussaillon !!!")
```

Objets en JavaScript

f) langage basé sur les prototypes



```
> professeur
< ▼ Personne {nom: "Tournesol", prenom: "Tryphon", profession: "savant fou", parler: f}
  ▶ parler: f (phrase)
    nom: "Tournesol"
    prenom: "Tryphon"
    profession: "savant fou"
  ▼ __proto__:
    ▶ aurevoir: f (phrase)
    ▶ constructor: class Personne
    ▶ __proto__: Object
> professeur.aurevoir("Un peu plus à l'ouest...")
```