

Fiche-Triche pour JavaScript

Julien Grange

<julien.grange@lacl.fr>



Généralités

JavaScript a été développé en 1995, dans le but premier de dynamiser les pages web. JavaScript était initialement uniquement exécuté côté client.

Caractéristiques

- Compilation Just-in-Time (autrefois interprété)
- Typage dynamique
- Simule l'orienté objet par prototypes
- Mémoire gérée par un ramasse-miettes
- Fonctionnel

Exécution

Du code JavaScript peut être exécuté côté client (dans un navigateur), ou côté serveur (via Node.js). Dans tous les cas, il est exécuté par un *JavaScript engine* - il en existe un dans chaque navigateur. Côté client, deux possibilités pour intégrer du code JavaScript dans du HTML :

- Pour les scripts de quelques lignes :

```
<script>
// code JavaScript
</script>
```
- Pour les scripts conséquents :

```
<script src="script.js"></script>
```

La première ligne d'un script doit toujours être "use strict";

Syntaxe élémentaire

Déclaration :	let a=0;	mutable
	const b=42;	non mutable
Affectation :	a="foo";	si a mutable
Commentaire :	// commentaire	
	/* commentaire sur	
	plusieurs lignes */	
Boîte de dialogue :	alert("Hello ");	
Print :	console.log("World !");	
Prompt :	let s=prompt("passwd:");	

Les ';' sont facultatifs, mais recommandés.

Une valeur spéciale pour représenter l'absence de valeur : **undefined**. Attention, tous les **undefined** sont égaux !

Nombres

Entiers et flottants sont représentés par le même type. La prudence est de mise concernant la division :

13/5 s'évalue à 2.6

Math.floor(13/5) s'évalue à 2

Trois valeurs spéciales :

- 42/0 s'évalue en **Infinity**
- -5/0 s'évalue en **-Infinity**
- "foo"+1 s'évalue en **NaN** (*not a number*), mais aussi
– NaN+3

- Infinity-Infinity
- Infinity*0

Pour tester ces valeurs, on utilise **isFinite()** et **isNaN()** : cette dernière est indispensable, puisque **NaN===NaN** s'évalue à **false** !

Conversion : **Number("42")**

Chaînes de caractères

Pas de notion de caractère : un caractère est une chaîne de taille 1.

Littéral : 'foo' ou "foo" ou `foo`

Formatage : `4 fois 5 font \${4*5}` (backticks)

Concaténation : "Hello " + "world!"

Caractère : "chaîne".at(2) "a"

"chaîne".at(-2) "n"

Sous-chaîne : "chaîne".slice(1,3) "ha"

"chaîne".slice(1) "haine"

Conversion : **String(42)**

Expressions booléennes

Deux booléens : **true** et **false**

Inégalité : **a<b** **a<=b**

Égalité : **a==b** **a!=b**

Égalité stricte : **a===b** **a!==b**

Conversion : **Boolean(42)**

Opérateurs : **!e** **e1 && e2** **e1 || e2**

L'évaluation est paresseuse : si **e1** est évaluée à **False** dans l'expression **e1 && e2**, alors **e2** ne sera pas évaluée.

Égalité

L'égalité stricte **===** requiert l'égalité des types, alors que **==** convertit en nombres si les types sont différents :

- **1==true** et **"042"==42.0** s'évaluent à **true**
- **1===true** et **"042"===42.0** s'évaluent à **false**

Inégalité

Sur les chaînes de caractères, **<** compare selon l'ordre lexicographique : **"10"<"2"** est évalué à **true**.

Conversion

Conversion automatique dans une conditionnelle ou une boucle **while**, ou conversion manuelle via **Boolean()**.

- 0 et NaN sont les seuls nombres convertis en **false**
- "" est la seule chaîne de caractères convertie en **false**
- **undefined** est converti en **false**

Structures de contrôle

Conditionnelle

```
if (cond1){
// bloc if
} else if (cond2){
// bloc else if
} else {
// bloc else
}
```

Notation ternaire :

```
cond ? expr.true : expr.false;
```

Boucle for

```
for (let i=0; i<10; i++){
// boucle for i=0,1,2,...,9
}
```

Boucle while

```
while (cond){
// bloc while
}
```

Fonctions

Définition

Il existe trois manières de définir une fonction :

1. Déclaration de fonction :

```
function multiplie(a,b){
return a*b;
}
```
2. Expression fonctionnelle :

```
let multiplie = function (a,b){
return a*b;
};
```
3. Lambda expression :

```
let multiplie = (a,b) => a*b; pas de return
```

Attention : ne jamais utiliser pour coder une méthode !

Les fonctions sont citoyennes de premier ordre : ce sont des valeurs comme les autres.

```
let f = multiplie;
f(2,3); // s'évalue à 6
```

Arguments

Si l'on appelle une fonction avec trop d'arguments, les arguments excédentaires sont ignorés. S'il manque des arguments, ils prennent la valeur **undefined**.

On peut définir des valeurs par défaut pour les arguments :

```
function multiplie(a,b=2){
return a*b;
}
multiplie(3); // s'évalue à 6
multiplie(3,4); // s'évalue à 12
```

Timeout

Pour appeler une fonction **f** dans **n** millisecondes :

```
setTimeout(f,n);
setTimeout renvoie un identifiant, qui peut être passé à clearTimeout pour déprogrammer l'appel.
```

Tableaux

Déclaration :	let t = [0,"un",2];	
Longueur :	t.length	3
Élément :	t[1]	"un"
Tranche :	t.slice(0,2)	[0,"un"]
Recherche d'indice :	t.indexOf("un")	1
Ajout en queue :	t.push(3)	
Retrait en queue :	t.pop()	retire&renvoie 3

Itération

Pour boucler sur les éléments d'un tableau :

```
for (let elt of t){
  // elt = t[0],t[1],...,t[t.length-1]
}
```

Pour appliquer la fonction `f` à chaque élément de `t` :

```
t.forEach(f);
```

Pour construire le tableau obtenu en appliquant la fonction `f` à chaque élément de `t` :

```
let tmap = t.map(f);
[3,5,2,8].map( n => 2*n ); // [6,10,4,16]
```

Pour construire le tableau obtenu en gardant uniquement les éléments de `t` sur lesquels la fonction booléenne `f` renvoie `true` :

```
let tfilter = t.filter(f);
[3,5,2,8].filter( n => n>3 ); // [5,8]
```

Pour appliquer une fonction `f` sur les éléments d'un tableau `t`, de gauche à droite, en accumulant les résultats partiels :

```
let a = t.reduce(f,init);
[3,5,2,8].reduce(
  (acc,n) => acc+n+"/" , "/"); // "/3/5/2/8/"
```

Objets et classes

Objets

Format `clef: valeur`, où `clef` est une chaîne de caractères :

```
let mouton = {
  appellation: "Pauillac",
  millesime: 2009,
  stock: 12
};
```

Lecture :

<code>mouton["stock"]</code>	s'évalue à 12
<code>mouton.stock</code>	synonyme (sans "")
<code>mouton.prix</code>	s'évalue en <code>undefined</code>

Écriture :

Appartenance : `"stock" in mouton` s'évalue à `true`

Pour boucler sur les clefs d'un objet :

```
for (let clef in mouton){
  // clef = "appellation", "millesime", "stock"
}
```

En affectant une expression fonctionnelle à une clef, on déclare une méthode. On peut alors utiliser `this` pour faire référence à l'objet.

```
let chien = {
  motif: "ouaf",
  aboie: function() {
    return this.motif + this.motif;
  }
};
alert(chien.aboie()); // "ouafouaf"
```

Prototype

Champ d'un objet, qui peut contenir un objet ou `null`.

Si un attribut ou une méthode n'est pas trouvée dans un objet, alors on va les chercher dans son prototype.

Deux méthodes pour accéder au prototype :

```
Lecture : Object.getPrototypeOf(obj)
Écriture : Object.setPrototypeOf(obj,proto)
Les prototypes permettent l'héritage :
let bouledogue = {
  motif: "rhhh"
};
Object.setPrototypeOf(bouledogue,chien);
alert(bouledogue.aboie()); // "rhhhrhhh"
```

Classes

```
class Etudiant {
  constructor(nom,numetu) {
    this.nom = nom;
    this.numetu = numetu;
  }
  toString() {
    return `${this.nom}, numéro ${this.numetu}`;
  }
}
```

À chaque appel `new Etudiant(nom,numetu)` :

1. un objet vide est créé et lié à `this`
2. `constructor(nom,numetu)` est appelée
3. on attribue à `this` comme prototype un objet contenant toutes les méthodes de `Etudiant`
4. `this` est renvoyé

En réalité, on peut appeler `new f()` pour n'importe quelle fonction `f` ! Dans ce cas, `f.prototype` (à ne pas confondre avec `Object.getPrototypeOf(f)`...) devient le nouveau prototype du `this` renvoyé.

Une classe n'est donc rien d'autre qu'une fonction (dont le code est celui de son `constructor`), possédant un champ `prototype` qui contient toutes les méthodes de la classe.

Pour tester l'appartenance à une classe :

```
obj instanceof Etudiant;
```

Pour hériter d'une classe :

```
class EtuInfo extends Etudiant{
  constructor(nom, numetu, os){
    super(nom,numetu)
    this.os = os;
  }
  ...
}
```

Mot-clef `this`

En JavaScript, `this` est lié selon des règles qui varient en fonction du contexte.

Hors d'une fonction

`this` fait référence à l'objet global, à savoir

- `window` dans un navigateur
- `global` dans `Node.js`

Dans une fonction

```
function f() {
  return this;
}
```

La valeur de `this` n'est pas déterminée lors de la déclaration de la

fonction ; `this` est lié à chaque appel.

- Si l'appel est de la forme `obj.f()`, `this` est lié à `obj` :

```
let obj = {};
obj.f = f;
console.log( obj.f() ); // obj
```
- Si l'appel est de la forme `f()`, `this` est lié à `undefined` :

```
let g = obj.f;
console.log( g() ); // undefined
```

Lier `this` autrement : `bind` et `call`

Si l'on cherche à maîtriser la valeur de `this`, on peut utiliser `f.bind()` :

```
let obj2 = {};
let h = obj.f.bind(obj2);
console.log( h() ); // obj2
```

On peut utiliser `f.call(o,a1,...)` pour évaluer `f(a1,...)` où `this` est lié à `o`.

Dans une lambda expression

`this` est lié au moment de la déclaration, et ne change plus par la suite. On ne peut donc pas utiliser `bind` et `call`.

```
let f = () => this;
console.log( f() ); // window
let obj = {};
obj.f = f;
console.log( obj.f() ); // window
```

De ce point de vue,

```
() => ... rappelle function () {...}.bind(this)
```

Promise, async et await

Promise

Une instance de la classe `Promise` représente une tâche à exécuter de manière asynchrone :

```
let promise = new Promise( function(resolve, reject) {
  ...
  resolve(result) ou reject(error)
});
```

Les fonctions `resolve` et `reject` sont fournies ; on appelle `resolve` en cas de réussite de la tâche, et `reject` en cas d'erreur.

On peut demander à exécuter un callback suite à la réalisation de la tâche, de manière non bloquante :

```
promise.then(
  (result) => ...,
  (error) => ...
);
```

Le premier argument de `then` est le callback appelé en cas de réussite, le second est le callback d'erreur.

async et await

On peut déclarer des fonctions asynchrones qui enrobent leur résultat dans une `Promise` :

```
async function f() {
  ...
  return result;
}
```

qui est l'équivalent (plus lisible) de

```
function f() {
  return new Promise( (resolve, reject) => {
    ...
    resolve(result);
  });
}
```

Dans une fonction `async`, on peut attendre le résultat d'une `Promise` – et donc d'une autre fonction `async` – grâce à `await` :

```
async function g() {
  let resultPromise = await promise;
  ... // exécuté une fois que promise est achevée
  let resultF = await f();
  ... // exécuté une fois que f retourne
}
```

Lorsqu'on bloque dans un `await`, le code d'autres fonctions `async` et/ou celui du corps du script continue de s'exécuter.

JavaScript et HTML

Chaque balise HTML est représentée en un élément du DOM (Document Object Model). Il s'agit d'un objet JavaScript, qui possède un champ pour chaque attribut de la balise associée, ainsi que des champs supplémentaires qui permettent d'accéder aux enfants, au contenu de la balise, etc.

Les éléments forment un arbre, dont la racine est `document`.

Recherche d'élément

Pour trouver un élément, plusieurs possibilités :

- `document.getElementById(id)`
- `elem.querySelectorAll(css_query)`
- `elem.querySelector(css_query)` (premier élément)

où les `css_query` de base sont de la forme

```
tag          balise tag
.c           classe c
#i          id i
[attr]      possédant un attribut attr
[attr="foo"] possédant un attribut attr de valeur "foo"
```

et peuvent se combiner entre eux, notamment via

```
s1 s2      s2 descendant de s1
s1 > s2    s2 enfant de s1
```

Navigation

```
parent de elem :      elem.parentElement
collection des enfants de elem : elem.children
frère précédant elem : elem.previousElementSibling
frère suivant elem :  elem.nextElementSibling
```

Ajout d'éléments

`document.createElement(tag)`, où `tag` est le nom de la balise.

Pour insérer `newElem`

```
comme premier fils de elem : elem.prepend(newElem)
comme dernier fils de elem : elem.append(newElem)
comme frère précédant elem : elem.before(newElem)
comme frère suivant elem : elem.after(newElem)
à la place d'elem : elem.replaceWith(newElem)
```

Modification du contenu

La lecture et l'écriture se font via l'attribut `innerHTML` :

```
lecture : elem.innerHTML
écriture : elem.innerHTML = "foo";
```

Pour un bouton, on utilisera à la place l'attribut `value`.

Accès aux attributs

Chaque attribut de la balise correspond à un champ de l'élément, du même nom. Par exemple, pour modifier le champ `href` d'une balise, on écrira

```
elem.href = "url.html"
```

Cas particulier : l'attribut `class` est accessible via les champs `className` et `classList`.

La valeur de l'attribut `style` est lui-même un objet, dont les clés sont les identifiants CSS habituels, écrits en camel-case. Par exemple, pour passer en rouge l'arrière-plan d'un élément ("`background-color`" en CSS), on écrira

```
elem.style.backgroundColor = "red"
```

Enregistrement de callback

Pour déclencher le callback `f` à chaque clic sur `elem` :

```
elem.addEventListener(event, f);
```

où `event` peut, entre autres, prendre comme valeur

```
click      déclenchement sur un clic
dblclick   déclenchement sur un double-clic
contextmenu déclenchement sur un clic droit
mouseover  déclenchement au passage de la souris
```

Si on ajoute comme troisième argument `once=true`, déclenchement seulement à la première occurrence.

Pour retirer ce même callback :

```
elem.removeEventListener(event, f);
```