Introduction à JavaScript

Alexandre Niveau

GREYC — Université de Caen

En partie adapté des cours de Jean-Marc Lecarpentier et Hervé Le Crosnier

Récapitulatif (1)

- Les pages web sont en HTML/CSS
- Un navigateur web est (par définition) un logiciel qui :
  - sait afficher du HTML/CSS
  - sait communiquer en HTTP pour récupérer les pages auprès d'un serveur
- HTML n'est pas un langage de programmation : pas d'instructions, pas de conditions, pas de boucles...
- Langage **statique**

Récapitulatif (2)

- Pour créer du contenu dynamique (qui dépend d'une BD, des requêtes de l'utilisateur, etc.), on utilise un vrai langage de programmation qui va **générer du HTML**
- Dans ce cours, on utilise le langage PHP
- C'est le serveur qui exécute le PHP, ce qui génère une page HTML qui est ensuite renvoyée au client
- **Ça ne change rien pour le client :** il récupère une page HTML *statique* dans tous les cas
- Pour que la page soit modifiée, il faut faire une nouvelle requête HTTP

Limites du modèle

- Impossible de rendre la page interactive
- On voudrait pouvoir afficher/cacher du contenu, zoomer sur des images, vérifier un formulaire en temps réel, etc.
- C'était bien pire en 1995 : pas de CSS, encore moins de CSS3 avec ses pseudo-classes, ses media queries et ses animations

Solution

- Le navigateur doit pouvoir exécuter des programmes présents dans les pages web

- Ces programmes doivent pouvoir modifier le contenu de la page (ajouter des éléments HTML, changer le CSS…)
- Ces programmes doivent être lancés lors de certaines actions de l'utilisateur (clics, scroll, survol…)
- Le langage utilisé pour écrire ces programmes est JavaScript.

JavaScript

- Conçu par Brendan Eich chez Netscape en une dizaine de jours en 1995
- Appelé d'abord « LiveScript » puis « JavaScript » pour des raisons marketing
- Sorti avec Netscape 2.0 en mars 1996 ; grand succès, donc Microsoft se dépêche d'implémenter sa version (JScript) pour IE 3.0 sorti en août
- Netscape propose très rapidement (fin 1996) à l'organisme Ecma de standardiser le langage

ECMAScript

- Standardisé par l'Ecma en 1997 sous le nom d'ECMAScript (compromis pour éviter les problèmes de copyright)
- ECMAScript est une norme, JavaScript et JScript en sont des implémentations (c'est aussi le cas de l'ActionScript de Flash)
- Dernière version de la norme : ECMAScript 7.0 [http://www.ecma-international.org/ecma-262/7.0/], en 2016…
- … mais la version 6 est encore en cours d'adoption (tout ne marche pas partout !) voir les nouveautés de la version 6 [http://es6-features.org/]
- Seul le *langage* est standardisé par l'Ecma, pas son interaction avec le navigateur et le HTML, qui est standardisé par le W3C.

Caractéristiques de JavaScript

- La syntaxe ressemble vaguement au Java, mais les concepts de base n'ont rien à voir
- Langage interprété, typage dynamique, orienté objet, fonctionnel
- Dépend fortement de l'environnement d'exécution : par exemple le langage n'a pas d'entrées/sorties standard ou de mécanisme d'importation
- L'environnement d'exécution typique est le navigateur web, qui permet d'utiliser
  - `alert("Texte")` pour afficher du texte dans une fenêtre pop-up
  - `console.log("Texte")` pour afficher du texte dans la console (dans Firefox, « Web developer ⇒ Web Console », ou Ctrl+Shift+K)
  - `prompt("Question")` ouvre un pop-up demandant à l'internaute de remplir un champ de texte, et renvoie le résultat (sous forme de chaîne)
  - `confirm("Question")` ouvre un pop-up demandant à l'internaute de confirmer ou d'annuler, et renvoie `true` ou `false` en fonction de son choix

Utiliser un script dans une page HTML

- On peut placer le script directement dans la page, dans un élément `<script>` :

```
<script>
    alert("Salut !");
</script>
```

- On peut aussi exécuter un script contenu dans un fichier (« script externe ») :

```
<script src="URL du script externe"></script>
```

  Utile dans les cas suivants :

  - Script long
  - Utilisation de scripts écrits par d'autres (librairies, frameworks)
  - Partage de fonctions entre plusieurs pages HTML

[#ou-placer-script]
Où placer l'élément `script` ?

- L'élément `<script>` peut se placer n'importe où dans le document
- Sa place « logique » est dans l'en-tête (élément `<head>`), puisqu'il ne fait pas partie du contenu
- Il y a cependant deux problèmes :
  - lorsque le parseur HTML rencontre un élément `script`, il le télécharge (si nécessaire) puis l'exécute, et *attend* qu'il soit terminé avant de reprendre le rendu du HTML

    ➤ si le script met longtemps à être téléchargé ou exécuté, l'internaute ne voit rien pendant ce temps
  - on a souvent besoin de manipuler des éléments HTML de la page dans les scripts ; il faut qu'ils aient déjà été créés par le parseur HTML
- La solution classique : mettre l'élément `script` à la fin de la page, juste avant de fermer le `body`.
- Meilleure solution (pour les scripts externes à la page) : utiliser l'attribut `defer`.

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

  Le parseur ne s'arrête pas pour télécharger et exécuter `monscript.js` ; le fichier est téléchargé en parallèle du parsage et exécuté à la fin.
- Plus de détails sur `defer` (et sur son petit frère `async`) dans cet article [http://www.growingwiththeweb.com/2014/02/async-vs-defer-attributes.html]

Variables

- On déclare une variable avec `let` :

```
let toto = 4;
alert(toto);
```

```
toto = "bonjour";
alert(toto);
```

- Variables dynamiques, non typées (comme en Python ou PHP)
- Les noms de variables peuvent contenir des caractères alphanumériques et les caractères underscore et dollar
- Noms sensibles à la casse. On utilise la notation *camelCase* par convention
- **Attention** : `let` est un ajout relativement récent, vous trouverez de nombreux tutoriels et exemples les variables sont déclarées avec `var`. La gestion de la portée des variables est alors différente.

  ➤ ne pas utiliser `var`, sauf si vous avez une bonne raison (compatibilité avec de vieux navigateurs). *Interdit dans le cadre de ce cours*.

Strict mode

- Pas mal de pièges en JS peuvent être évités en utilisant le « mode strict » introduit dans ECMAScript 5
- Syntaxe: `"use strict";` au début d'un script (ou d'une fonction) — les guillemets font partie de la syntaxe (la ligne est simplement ignorée par les moteurs JS plus anciens)
- Empêche l'utilisation des aspects les plus dangereux (par exemple utiliser des variables non déclarées)
- Envoie des exceptions plus souvent
- A priori recommandé, sauf si vous savez ce que vous faites…

  ➤ *Obligatoire dans le cadre de ce cours*
- Quelques détails [https://www.nczonline.net/blog/2012/03/13/its-time-to-start-using-javascript-strict-mode/]

Séparation des instructions

- Les instructions sont séparées par des points-virgules…
- … mais dans la plupart des cas les points-virgules sont optionnels si un saut de ligne est présent
- Le code suivant est valide :

  ```
  let toto = 5
  let titi = 3
  alert(toto)
  alert(titi)
  ```

  En effet, le parseur détecte que le code n'aurait pas de sens sans point-virgule à la fin de chaque ligne.
- Si enlever le saut de ligne donne un code parsable, alors il faut absolument mettre un point virgule :

```
a = b + c
(d + e).print()
```

est interprété comme

```
a = b + c(d + e).print()
```

- Inversement, il faut garder en tête que contrairement à C ou Java, les sauts de ligne peuvent être significatifs :

```
return
  true;
// est interprété comme
return;
  true;
```

Points-virgules ou pas ?

- Avantage d'omettre les points-virgules optionnels : moins à écrire, moins à lire
- Avantage de toujours mettre tous les points-virgules : pas besoin de réfléchir aux règles d'insertion automatique
- … cela dit les règles peuvent être résumées simplement comme suit : il faut mettre un point-virgule si la ligne suivante commence par `[` ou `(`, ou un des opérateurs arithmétiques (`+`, `-`, `*`, `/`).
- Rules of Automatic Semicolon Insertion [http://www.ecma-international.org/ecma-262/7.0/index.html#sec-rules-of-automatic-semicolon-insertion] dans la spec
- Des articles arguant qu'il vaut mieux mettre des points-virgules partout : The Dangers of JavaScript's Automatic Semicolon Insertion [https://web.archive.org/web/20170803090524/http://cjihrig.com:80/blog/the-dangers-of-javascripts-automatic-semicolon-insertion/], et A Bit of Advice for the JavaScript Semicolon Haters [http://benalman.com/news/2013/01/advice-javascript-semicolon-haters/].
- Des articles arguant que ce n'est pas la peine : No, you don't need semicolons [https://medium.com/@goatslacker/no-you-dont-need-semicolons-148d936b9cf2], et Semicolons in JavaScript are optional [https://mislav.net/2010/05/semicolons/]

Types de base

- Nombre (pas de distinction entre entier et flottant)
- Booléen : `true` et `false`
- Chaîne de caractères : `"Toto"` ou `'Toto'` (équivalent)
- Objet : voir plus loin. C'est le seul type dont les instances ne sont pas immutables.
- Symbol : un nouveau type (introduit dans ES6) fait pour représenter des identifiants (doc MDN [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Symbol])
- `null` : destiné à être utilisé pour dire « aucune valeur » (doc MDN [https://

developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/null])

- `undefined` : c'est ce que contiennent les variables n'ayant pas eu de valeur assignée
- `typeof maVariable` permet d'obtenir le type du contenu de maVariable (doc MDN [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/L_op%C3%A9rateur_typeof], attention `typeof null` renvoie `"object"` pour des raisons historiques...)

Opérateurs de base

- Les opérateurs sont en gros ceux du Java (et donc du C)
- Pour les chaînes, concaténation avec +, conversion implicite (typage faible) : `"Toto a " + 45 + " ans"` (il suffit qu'une seule opérande soit une chaîne pour que tous les + soient des concaténations)
- Attention à la conversion implicite des chaînes :

```
"123" + 4  // "1234"
"123" - 4  // 119
"a" + 2 + 1  // "a21", pas "a3" !
```

Typage faible et comparaison

- JavaScript est faiblement typé (comme awk ou PHP)
- Attention, comme en PHP, == et != sont des opérateurs d'(in)égalité *faible* !

```
console.log("1" == 1);  // true
console.log("0" == false);  // true
console.log("" == false);  // true
console.log("" == "0");  // false
```

- Pour les chaînes de caractères notamment, penser à utiliser les opérateurs stricts : === et !==

Structures de contrôle de base

- Pareil que le Java (et donc le C)
- if...else, switch
- opérateur ternaire `cond? exprSiVrai: exprSiFaux`
- for, while, do...while

```
for (let i = 0; i < 20; i++) {
    console.log("itération numéro " + i);
}
```

Déclaration de fonctions

- Ressemblent à celles de PHP et awk : mot-clé `function`, pas d'indication du type de retour

```
function toto(arg1, arg2) {
  alert(arg1 + ' ' + arg2);
  return arg1;
}
```

- Attention : même en mode strict, les fonctions peuvent être appelées avant leur déclaration, et peuvent être déclarées plusieurs fois (c'est la dernière déclaration qui gagne)
- Appeler une fonction en ajoutant des paramètres ou en les omettant est autorisé (les paramètres en trop sont ignorés ; les paramètres non fournis sont `undefined`)
- Dans une fonction :
  - les paramètres et les variables déclarées dans la fonction sont locales, comme c'est le cas en général
  - *les variables non déclarées dans la fonction sont globales* !

Démo portée des variables dans les fonctions [demo/portee2.html]

Portée des variables

- Les variables déclarées avec `let` sont *locales à leur bloc* (comme en Java ou en C) : pas de problème particulier
- Ce n'est pas le cas pour les variables déclarées avec `var` :

```
var toto = "youpi";
console.log(toto);  // affiche youpi
if (true) {
        var toto = "youpa";
        console.log(toto);  // affiche youpa
}
console.log(toto);  // affiche youpa
```

Résultat avec var et let [demo/portee1.html]

➤ **Ne pas utiliser `var` !**

Objets

- Un seul type mutable en JavaScript : les « objets »
- Les objets en JavaScript ressemblent aux tableaux associatifs de PHP/awk et aux dictionnaires de Python
- Un objet est un ensemble de couples clé-valeur

```
let toto = { nom: "Durand", prenom: "Toto", age: 45 }
```

- Différentes façons d'accéder aux valeurs :

```
alert("Je m'appelle " + toto.prenom +
```

```
        " et j'ai " + toto["age"] + " ans");
```

Tableaux

- Les tableaux sont des objets particuliers, qui ont un attribut `length`

```
let fruits = ['pomme', 'poire', 'abricot'];
console.log(fruits[0]);
console.log(fruits.length);
fruits[100] = 'ananas';
console.log(fruits.length);
```

- Ajouter un élément au tableau : `fruits.push('banane')`

démo [demo/tab.html]

- Liste de toutes les méthodes disponibles sur les tableaux [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array]

Manipulation des fonctions

- En JavaScript les fonctions sont des « entités de première classe » [https://en.wikipedia.org/wiki/First-class_citizen]
- On peut mettre une fonction dans une variable, ou la passer en paramètre d'une autre fonction :

```
function appelerNFois(n, f) {
  for (let i = 0; i < n; i++) {
    f();
  }
}
```

- Très souvent utilisé : dans le contexte du web, on a souvent besoin de passer en paramètre des fonctions appelées *callbacks*, par exemple pour les *listeners* d'événements.
- Exemple simple : fonction `setTimeout`, qui prend en argument une fonction et une durée, et exécute la fonction quand la durée est écoulée

```
function afficherSurprise() {
        console.log("Surprise !!!");
}
setTimeout(afficherSurprise, 5000);
```

Résultat [demo/timeout.html]

Fonctions anonymes

- On peut même utiliser le mot-clef `function` comme un *opérateur* :

```
toto = function coucou() {
  alert("coucou");
```

```
};
coucou();
toto();  // même effet que coucou()

titi = function () {
  alert("coucou anonyme");
};
titi();
```

Le deuxième exemple montre que le nom d'une fonction est optionnel dans ce cas. On parle de fonction anonyme.

- Souvent utilisé justement pour les callbacks : plutôt que de déclarer une fonction qui n'est utilisée qu'une fois, on la crée à la volée

```
setTimeout(function () {
        console.log("Surprise !!!");
}, 5000);
```

Renvoyer une fonction

- Une fonction peut retourner une fonction :

```
function creerAfficheur(intro) {
  return function (texte) {
    console.log(intro + " : " + texte);
  };
}

let presenter = creerAfficheur("Je vous présente");
presenter("Toto");
presenter("Titi");

let accuser = creerAfficheur("J'Accuse");
accuser("Martine Durand");
```

Résultat [demo/closure1.html]

Clôtures

- Ça devient très intéressant si la fonction renvoyée utilise des variables locales à la fonction englobante :

```
function creerCompteur() {
  let i = 0;
  return function () {
    console.log(i);
    i++;
  };
}

compteur = creerCompteur();
compteur();  // affiche 0
compteur();  // affiche 1
compteur2 = creerCompteur();
compteur2();  // affiche 0
compteur();  // affiche 2
```

- La fonction renvoyée garde une référence vers les variables locales à la fonction englobante. On parle de *clôture* (*closure*). Ça peut paraître étrange, mais c'est plus intuitif qu'il n'y paraît.

Le piège des clôtures avec `var`

- Le fait qu'une clôture garde une référence vers les variables peut être piégeant si on utilise `var`
- L'erreur classique :

```
var mes_fonctions = [];
for (var i = 0; i < 10; i++) {
        mes_fonctions[i] = function () {
                console.log("Je suis la fonction numéro " + i + " !");
        };
}
var toto = mes_fonctions[8];
toto();
```

  Qu'affiche l'appel à la fonction `toto()` ?

Le problème ne se pose pas avec `let` : s'il est utilisé pour déclarer l'indice dans un `for`, lui donne un nouveau *binding* à chaque itération, précisément pour éviter le problème en question
➤ **Ne pas utiliser `var`**

- Si on ne peut pas utiliser `let`, les solutions classiques sont :
  - utiliser un constructeur de fonctions (comme expliqué ici [http://conceptf1.blogspot.fr/2013/11/javascript-closures.html])
  - forcer un scope local dans la boucle en utilisant une fonction anonyme invoquée immédiatement (IIFE) [http://tobyho.com/2011/11/02/callbacks-in-loops/], ce qui nécessite une syntaxe particulière [https://blog.mariusschulz.com/2016/01/13/disassembling-javascripts-iife-syntax]
  - utiliser plutôt la méthode `forEach` des tableaux [https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/forEach] (si ce sont les éléments d'un tableau que l'on parcourt)

Débugger avec `console.log`

- `console.log` ne se contente pas d'afficher du texte : si on lui passe un objet, il en affichera une représentation interactive dans la console
- Attention cependant : quand on interagit avec cette représentation, c'est **la version courante de l'objet** que l'on voit, pas l'objet tel qu'il était au moment où `console.log` a été exécuté !

  ➤ `console.log` manipule des *références* vers les objets, pas des copies
- Cela peut poser des problèmes lors du debug, car l'objet peut avoir changé…

Démo [demo/consolelog.html]

- Pour afficher un objet tel qu'il est à un moment précis, il faut le convertir en chaîne de caractères.

- Une façon simple est de le sérialiser, par exemple avec `JSON.stringify(toto)` : tout ne sera pas converti [https://flaviocopes.com/how-to-clone-javascript-object/#json-serialization], mais pour débugger ça peut suffire
- Si l'objet est très gros, ce n'est pas pratique ; une possibilité est d'afficher un objet reconstruit à partir de la sérialisation, avec `JSON.parse(JSON.stringify(toto))`.

  ➤ on bénéficie de l'affichage interactif, sans que l'objet n'ait été mis à jour. (mais on hérite des limites de `JSON.stringify`)

Démo [demo/consolelog2.html]

POO en JavaScript

- JavaScript permet de faire de l'OO sans classes, grâce au mot-clef `this`, qu'on ne détaillera pas dans ce cours
- Introduction to Object-Oriented JavaScript [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript] sur MDN
- On s'appuie sur le fait que les fonctions sont des objets
- Héritage particulier, basé sur le chaînage de prototypes [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain]
- ES6 fournit du sucre syntaxique pour créer des objets plus simplement (avec le mot-clef `class`). Pas encore utilisable partout. Un tuto (avec aussi des explications détaillées sur le fonctionnement des prototypes [https://scotch.io/tutorials/better-javascript-with-es6-pt-ii-a-deep-dive-into-classes]

Spécifications et normes

- Norme ECMAScript 5.1 de 2011, utilisable partout [http://www.ecma-international.org/ecma-262/5.1/]
- Norme ECMAScript 6.0 de 2015, encore en cours d'adoption [http://www.ecma-international.org/ecma-262/6.0/]
- Norme ECMAScript 7.0 de 2016 [http://www.ecma-international.org/ecma-262/7.0/]

Références et guides

- Javascript sur le Mozilla Developer Network [https://developer.mozilla.org/en-US/docs/JavaScript]
- JavaScript Garden [http://bonsaiden.github.io/JavaScript-Garden] documentation sur les pièges du JS
- A Survey of the JavaScript Programming Language [http://javascript.crockford.com/survey.html]

Tutoriels

- Tuto Javascript sur codecademy.com [http://www.codecademy.com/tracks/

javascript]

- Eloquent JavaScript [http://eloquentjavascript.net/]
- async vs defer attributes [http://www.growingwiththeweb.com/2014/02/async-vs-defer-attributes.html]
- JS inheritance and the prototype chain [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain]

Lectures complémentaires

- Douglas Crockford on JavaScript [http://javascript.crockford.com/]
- The dangers of JavaScript's automatic semicolon insertion [http://cjihrig.com/blog/the-dangers-of-javascripts-automatic-semicolon-insertion/]
- A Bit of Advice for the JavaScript Semicolon Haters [http://benalman.com/news/2013/01/advice-javascript-semicolon-haters/]
- Introduction to Object-Oriented JavaScript [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript]
- Better JavaScript with ES6, Pt. II: A Deep Dive into Classes [https://scotch.io/tutorials/better-javascript-with-es6-pt-ii-a-deep-dive-into-classes]

[http://creativecommons.org/licenses/by-nc-sa/4.0/]