

# Le JavaScript moderne

Par artragis  
et Cygal



[www.openclassrooms.com](http://www.openclassrooms.com)

*Licence Creative Commons 7 2.0  
Dernière mise à jour le 3/06/2012*

## Sommaire

Sommaire .....	2
Lire aussi .....	1
Le JavaScript moderne .....	3
L'histoire mouvementée de JavaScript .....	3
La préhistoire .....	3
Redécouverte de JavaScript .....	4
Normalisation .....	4
L'assembleur du web .....	5
Cas d'utilisation : le Site du Zéro .....	5
Applications web .....	6
Compilateurs source à source .....	7
Modules JavaScript .....	8
RequireJs .....	10
Pourquoi se limiter au navigateur ? .....	10
Le retour des modules .....	12
Un langage moderne .....	12
ECMAScript Harmony .....	14
Modules .....	14
Templates .....	14
Quelques autres nouveautés .....	16
Sources .....	17
Partager .....	17

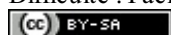
# Le JavaScript moderne



Par [Cygal](#) et [artragis](#)

Mise à jour : 03/06/2012

Difficulté : Facile  Durée d'étude : 1 heure, 30 minutes



1 visites depuis 7 jours, classé 23/807

Ces dernières années, l'utilisation de JavaScript a évolué jusqu'à en faire un **incontournable du web**. L'écrasante majorité des navigateurs supportent JavaScript, que ce soit pour les ordinateurs de bureau ou les téléphones portables. Les navigateurs web optimisent continuellement leurs moteurs JavaScript et les développeurs en profitent pour apporter des applications entières sur le web. Le succès de JavaScript est tel qu'il s'étend désormais à d'autres usages :

- [Node.js](#), [CouchDB](#) ou encore [Meteor](#) utilisent JavaScript pour créer des sites web, mais cette fois aussi du côté serveur ! Un seul langage est utilisé, ce qui facilite le développement.
- Les applications de bureau ne sont pas en reste : JavaScript fera partie des langages supportés pour écrire des applications dans [Windows 8](#) et [Qt 5](#), deux mastodontes pour les interfaces utilisateur de demain.

Paradoxalement, la réputation de JavaScript a longtemps été mauvaise, et on n'imagine pas forcément au premier abord toutes les possibilités aujourd'hui offertes par JavaScript. Dans ce contexte, tout programmeur a intérêt à se pencher sur JavaScript pour comprendre ses origines, savoir ce qui est possible aujourd'hui et ce qu'on peut espérer du langage dans le futur. C'est dans cette optique que se place cette article : nous espérons qu'elle puisse vous être utile en tant que programmeur et/ou décideur pour mieux comprendre ce que peut vous apporter ce langage **aujourd'hui et dans les années qui viennent**.

L'utilisation principale de JavaScript reste aujourd'hui l'aide à la **navigation sur un site web**. L'idée est de proposer aux utilisateurs ayant activé JavaScript des petites fonctionnalités pratiques rendant la navigation plus rapide et plus confortable sans pour autant impacter les utilisateurs ne disposant pas de JavaScript (tels que les moteurs de recherche). Cette façon d'utiliser JavaScript s'inscrit dans le cadre plus général du *progressive enhancement*. On se contente initialement de HTML pour apporter les fonctionnalités de bases aux clients les plus basiques : moteurs de recherche, lecteurs d'écrans, navigateurs texte. On conçoit ensuite la présentation en CSS avec des techniques de **responsive design** : on pense d'abord aux petits écrans des smartphones, puis le rendu est adapté aux écrans plus larges via les [media queries CSS3](#).

JavaScript permet ensuite de peaufiner l'expérience utilisateur par petites touches : nous prendrons pour exemple le Site du Zéro qui a toujours [suivi cette philosophie](#). Par exemple, la *speedbarre* située en haut de chaque page permet d'accéder rapidement aux divers contenus du site : l'utilisation de JavaScript améliore ici l'ergonomie en respectant mieux la [règle des trois clics](#). Nous ferons plus tard dans l'article un état des lieux plus complet de l'utilisation de JavaScript sur le Site du Zéro, qui se cache parfois où on ne l'attend pas.

Sommaire du tutoriel :



- [L'histoire mouvementée de JavaScript](#)
- [L'assembleur du web](#)
- [ECMAScript Harmony](#)

## L'histoire mouvementée de JavaScript

Il est important de réaliser que JavaScript et les bibliothèques telles que jQuery qui ont fait son succès ne sont pas apparues du jour au lendemain. Au contraire ! Cette section explique le long chemin parcouru depuis les années 1990.

### La préhistoire



C'est en 1995 qu'a eu lieu sur le web le début de la guerre des navigateurs. Netscape Navigator 1 et Internet Explorer 1 étaient les deux navigateurs les plus populaires, et chacun se battait pour grappiller des parts de marché à l'autre. Une des stratégies les plus utilisées à l'époque était d'ajouter des **améliorations** propriétaires puis d'encourager les webmasters à s'en servir. Les sites web ainsi créés ne fonctionnant pas aussi bien chez le concurrent, les utilisateurs étaient ainsi encouragés à installer le navigateur prévu par le webmaster. Naturellement, dès qu'une fonctionnalité devenait populaire, le concurrent l'implémentait aussitôt, et le cycle pouvait

recommencer.

C'est dans ce contexte que Brendan Eich a été recruté par Netscape, toujours en 1995. Sa mission était initialement d'**intégrer le langage fonctionnel Scheme** à Netscape 2 pour donner la possibilité à des personnes peu expérimentées d'**ajouter des effets à leurs pages**. La direction de Netscape a insisté pour que ce langage ressemble à Java, qui allait lui aussi être intégré pour rendre possible la création de composants plus sérieux, les applets Java. Le nom JavaScript viendrait d'ailleurs de la volonté d'**attirer l'attention des médias** sur ce nouveau langage, et d'entretenir la confusion avec Java. Néanmoins, ce sont deux langages très différents : Brendan Eich déclare s'être principalement inspiré de Scheme et Self pendant les 10 jours qu'il a eu pour concevoir un prototype. Au-delà du langage en lui-même, c'est à cette époque qu'est apparu DOM 0 qui permet d'utiliser JavaScript pour interagir avec une page web. Par exemple, ce code permet d'afficher une popup à l'utilisateur lors d'un clic sur un lien :

Code : HTML

```
<a href="#clickme" onclick='alert("Vous avez cliqué !")'>Cliquez-moi
!</a>
```

Au fil des années, de **nombreuses versions** de JavaScript sont apparues dans Netscape et Internet Explorer, apportant à chaque fois leur lot de nouveautés. À cette époque, l'utilisation de JavaScript dans des pages web s'appelait **DHTML**, et de nombreux **mauvais livres** en enseignaient les possibilités sans pour autant indiquer les bonnes pratiques. Les `alert()`, les curseurs personnalisés, les **menus déroulants inaccessibles** voire la **neige sur les pages web** étaient une plaie pour les utilisateurs. JavaScript était alors considéré comme un « **langage pour amateurs** ».

## Redécouverte de JavaScript

Ce n'est que depuis le milieu des années 2000 que les développeurs ont compris que JavaScript pouvait être utilisé judicieusement afin d'améliorer l'expérience de certains utilisateurs sans pénaliser les autres. C'est aussi à cette époque que l'objet XMLHttpRequest a été rendu disponible sur les principaux navigateurs, ouvrant la voie à des sites web plus dynamiques, capables de ne recharger qu'une partie des pages web : cette technique est connue sous le nom d'AJAX. **Ruby on Rails** proposait dès 2005 l'utilisation d'AJAX de **manière simple et transparente** ; de nombreux frameworks ont suivi par la suite.

Cependant, écrire du JavaScript restait réservé à une certaine élite du fait des différences entre les navigateurs. Afin de gommer ces différences, des bibliothèques telles que jQuery, Prototype ou encore Mootools deviennent disponibles en 2005/2006. Elles offrent notamment la possibilité de sélectionner des éléments de la page en utilisant de simples **sélecteurs CSS**. Par exemple, si je veux afficher tous les éléments cachés d'une page ayant la classe *secret*, il me suffit d'écrire

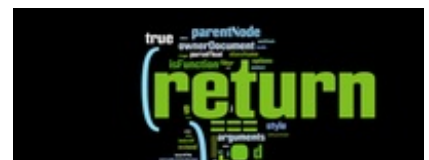
`$('.secret[display="none"]').show()`. Les apports de ces bibliothèques vont au-delà de ce simple exemple :

- un simple appel de fonction permet de récupérer des données du serveur via AJAX ;
- il est facile de réagir à des événements tel qu'un clic ou une soumission de formulaire ;
- de nombreuses animations tels que les fondus apportent des transitions plus douces ;
- et de **nombreuses autres fonctionnalités**.

Contrairement à ce qu'on pourrait croire, ces bibliothèques ne dispensent pas de connaître et comprendre les concepts JavaScript utilisés. Quoiqu'il en soit, ces bibliothèques ont ouvert la voie à une utilisation massive de JavaScript en facilitant grandement l'écriture de nombreux programmes allant du petit script à l'application complète.

## Normalisation

Pour s'assurer que JavaScript soit implémenté de la même manière dans tous les navigateurs, un **standard** spécifiant en détail le fonctionnement du langage est une condition nécessaire, même si pas forcément suffisante. Le comité TC39 existe au sein d'ECMA depuis 1996 pour réaliser ce travail d'équilibriste : il s'agit à la fois d'incorporer les extensions les plus pertinentes et de proposer les évolutions les plus prometteuses pour le langage, le tout devant former un ensemble cohérent. Le langage ainsi spécifié



se nomme **ECMAScript**. C'est d'ailleurs un abus de langage de parler de JavaScript qui n'est finalement que le nom de l'implémentation initialement proposée par Netscape et maintenant développée par Mozilla. Quand nous disons programmer en JavaScript, nous utilisons en fait une version spécifique d'ECMAScript.

Pendant longtemps, c'est ECMAScript 3 qui a été utilisé. Au début des années 2000, le comité s'est penché sur une mise à jour très ambitieuse : ECMAScript 4. Malheureusement, le langage était tellement complexe que les membres du comité n'ont pas réussi à se mettre d'accord et cette version a fini par être abandonnée. Les organisations n'étant pas favorables à ES4 (parmi lesquelles figuraient Microsoft, Google et Yahoo!) ont décidé de former leur propre comité pour proposer une mise à jour plus réaliste et moins ambitieuse. Le comité TC39 l'a finalement adoptée à la place d'ECMAScript 4, et c'est ainsi qu'ECMAScript 5 a été finalisé à la fin de l'année 2009.



Source: Infinities Loop

Parmi les fonctionnalités apportées par ES5 figurent des fonctions capables d'opérer sur des collections. Par exemple, `filter()` prend en paramètre une fonction et retourne une nouvelle collection contenant les éléments que la fonction a choisis de conserver :

#### Code : JavaScript

```
var ages = [12, 89, 23, 45, 5, 37];
var majeurs = ages.filter(function(age) { return age > 18; });
// Désormais, majeurs vaut [89, 23, 45, 37]
```

Le comité TC39 a décidé de ne pas s'arrêter avec ECMAScript 5 et travaille aujourd'hui sur ECMAScript Harmony, une nouvelle version prometteuse qui reprend certains éléments d'ECMAScript 4 et qui vise à faciliter l'utilisation de JavaScript dans des applications modernes. La dernière section de cette news vous présentera les nouveautés les plus intéressantes de cette nouvelle version d'ECMAScript.

## L'assembleur du web

JavaScript a principalement évolué en tant que langage de script pour les navigateurs. Cependant, son utilisation dépasse aujourd'hui ce cadre, et cette section se propose aussi de vous introduire à d'autres possibilités qui n'auraient pas été imaginables il y a quelques années.

Si vous êtes intéressés par ces technologies, sachez que [Nesquik69](#) et [Thunderseb](#) nous proposent deux tutoriels de qualité sur le Site du Zéro : [Dynamisez vos sites web avec Javascript !](#) et [AJAX et l'échange de données en JavaScript](#). Ils ne sont pas spécifiques à une bibliothèque telle que jQuery et constituent un excellent moyen de comprendre le fonctionnement de JavaScript tout en apprenant à l'utiliser au mieux : il est ensuite très facile d'utiliser jQuery, par exemple. Une autre excellente ressource est [JavaScript: The Good Parts](#) qui couvre en profondeur les bonnes pratiques à utiliser avec JavaScript.

## Cas d'utilisation : le Site du Zéro

Pour rendre cet article plus concret, nous avons décidé de montrer comment est utilisé JavaScript dans un vrai site web : le Site du Zéro. Ce n'est pas le seul site à le faire : Google Search, Wikipédia, [BaseCamp Next](#) ou encore [Trello](#) sont d'autres exemples de site tirant profit intelligemment des possibilités de JavaScript. Néanmoins, il est intéressant de proposer un « instantané » de l'utilisation du JavaScript sur un site web donné qui a toujours déclaré utiliser JavaScript pour [améliorer le confort](#) des utilisateurs, et non pour céder à une quelconque mode. Sur le Site du Zéro, une lecture du code JavaScript indique qu'il y a trois grandes utilisations :

- les publicités qui sont chargées en JavaScript après le reste de la page afin d'accélérer le chargement ;
- les outils de mesure (activité de l'utilisateur, performance de la page) ;
- les aides à la navigation.

Les **publicités** sont chargées une fois la page affichée pour ne pas ralentir l'utilisateur qui vient avant tout pour le contenu du site web : il n'y a rien de plus frustrant que d'attendre qu'une publicité s'affiche pour voir le contenu d'une page. Les publicités semblent être servies par Google AdSense, Smart AdServer ou parfois directement par le site pour le site : il s'agit alors d'auto-promotion pour des parties spécifiques du site (la boutique, par exemple). Il est intéressant de noter que le Site du Zéro n'impose pas les publicités à ses utilisateurs : il est possible de les [désactiver complètement](#) pour les utilisateurs qui le souhaitent. C'est exceptionnel sur le web et mérite d'être souligné.

Au niveau des **mesures**, il y a d'abord la performance côté client : diverses mesures dont le temps d'affichage sont effectuées via l'outil [Real User Monitoring](#) de New Relic. Cet outil permet aux développeurs de savoir ce qui prend du temps *en pratique* sur la page. Est-ce que c'est l'affichage des pubs ? Le téléchargement de jQuery ? Le traitement du JavaScript pour le navigateur ? C'est un outil précieux pour garder le site performant et rapide à utiliser.

Du point de vue de l'activité de l'utilisateur, une analyse des pages vues est faite via Google Analytics et XiTi. Au-delà du suivi des clics, l'activité sur chaque page est mesurée via [Chartbeat](#). Êtes-vous en train de bouger la souris ou de faire défiler la page ? C'est ce



Logo Chartbeat

qui indique à Chartbeat et au Site du Zéro que vous êtes en train de lire cet article. Si vous décidez de laisser un commentaire, les lettres tapées permettront de savoir que vous avez commencé à taper un message, avant même que vous ne décidiez de le soumettre. Ces statistiques ont de nombreuses utilités potentielles. Un exemple farfelu : elles pourraient être utilisées pour améliorer les articles. En effet, si tout le monde arrête sa lecture après ce passage, c'est peut-être qu'il était décourageant. 😊

Enfin, la plus grande partie du JavaScript écrit sur le Site du Zéro concerne les **aides à la navigation**. Elles sont nombreuses, mais jamais obscurives : il est possible d'utiliser les mêmes fonctionnalités sans JavaScript, c'est simplement moins agréable. Il y a plusieurs dizaines de telles fonctionnalités. Citons en quelques-unes :

- tous les formulaires du site sont équipées de la zForm, permettant de faciliter l'ajout de balises zCode permettant d'écrire du contenu plus riche (liens, titres, mise en forme, etc.),
- la sauvegarde automatique intégrée à la zForm permettant de ne pas perdre le contenu écrit suite à une fausse manipulation,
- le carrousel de la page d'accueil met en avant différents contenus les uns après les autres,
- les spoilers ne sont affichés en JavaScript qu'à la demande de l'utilisateur,
- l'auto-complétion lors de la recherche de membres,
- l'affichage de la timeline twitter ou du flux RSS des membres directement sous leur avatar,
- le calcul du prix total lors d'une commande de livres,
- ou encore l'affichage des avatars des auteurs d'un tutoriel au [survol de leur pseudo](#).

Certains ajouts ont une utilité limitée, d'autres comme la sauvegarde automatique sont rapidement devenus indispensables, mais rien de toute cela n'aurait été possible sans JavaScript.

## Applications web



Logo HTML5

Même si le Site du Zéro reste relativement conservateur dans son utilisation de JavaScript, il est possible de franchir un autre palier : écrire une véritable **application** avec JavaScript. La distinction entre une application web et un site web est [très floue](#), mais reste un moyen rapide de désigner des sites faisant énormément appel à JavaScript tels que GMail, la version classique de Twitter, ou encore Facebook. Étant donné que JavaScript est **le** langage que les navigateurs savent interpréter *rapidement*, ces applications peuvent se permettre d'utiliser de grandes quantités de JavaScript en sachant que le site restera rapide à l'utilisation. Différents moteurs tels que V8 ou SpiderMonkey

ont été améliorés au fil des années pour apporter le meilleur des techniques de compilation à la volée afin d'apporter des performances toujours meilleurs aux applications web. Cette performance a rendu possible l'écriture d'applications entières en JavaScript.

Pour écrire de tels applications, de nombreux frameworks web ont vu le jour, permettant aux développeurs de mieux s'organiser. [Backbone.js](#) a pour but de redonner de la structure au développement d'applications web en s'inspirant du [modèle MVC](#) très répandu dans les frameworks web tels que [Ruby on Rails](#), [Django](#) ou encore [Code Igniter](#). Plus récemment, [Meteorjs](#) et [Angularjs](#) ont fait parler d'eux pour l'approche novatrice qu'ils apportent au développement d'applications web en JavaScript. De tels frameworks sont un gros atout pour JavaScript : il n'a jamais été aussi simple d'écrire une application directement en JavaScript.

L'arrivée d'HTML5 est l'occasion pour le W3C de continuer à étendre les fonctionnalités accessibles à JavaScript, qui vont désormais bien plus loin que la manipulation de la page. Citons ici :

- l'accès à la position de l'utilisateur (jusque-là réservée aux applications natives) ;
- la présence d'un cache dans le navigateur, permettant à la fois d'utiliser des applications hors-ligne et de les démarrer plus rapidement ;

- canvas et WebGL pour des applications en 2D/3D toujours plus impressionnantes.

De plus en plus d'applications mobiles peuvent ainsi être développées directement en JavaScript, ce qui évite la lourdeur du développement des applications classiques : l'application fonctionne sur tous les smartphones, l'installation est aussi simple que la visite d'une page web, les mises à jour sont automatiques et la validation n'est plus nécessaire.

## Compilateurs source à source

De nombreux développeurs ne souhaitent pas attendre que la prochaine version de JavaScript soit disponible avant de commencer à l'utiliser. D'autres souhaitent un langage proche mais plus facile à écrire. D'autres encore peuvent considérer qu'un langage entièrement différent correspond mieux à leurs besoins ! Par exemple, si un zéro moyen souhaite écrire son application web en OCaml plutôt qu'en JavaScript, cela ne pose aucun problème : il suffit d'avoir un moyen de traduire le code OCaml en JavaScript ; par exemple `js_of_ocaml`.



Le principe est le même que pour un compilateur classique : on transforme un programme écrit dans un langage dans un autre langage. Par exemple, gcc est capable de transformer du C en langage assembleur adapté à un processeur donné. `js_of_ocaml` fait la même chose ; mais traduit de l'OCaml en JavaScript. C'est pour cette raison qu'on appelle JavaScript **l'assembleur du web**.



Le compilateur source à source le plus utilisé est certainement CoffeeScript. Sa particularité est de rester proche du JavaScript en ajoutant différentes formes de sucre syntaxique rendant le programme source plus concis.

Par exemple, il est possible en CoffeeScript d'écrire le code suivant :

### Code : JavaScript

```
arroseur = "Jacques"
arrose = "Martin"

[arroseur, arrose] = [arrose, arroseur]
```

Ce code permet d'échanger les deux variables sans passer par une variable temporaire. Le compilateur CoffeeScript produit alors ce code JavaScript qui sera exécuté sur le navigateur :

### Code : JavaScript

```
var arroseur = "Jacques";
var arrose = "Martin";

var _ref;
_ref = [arroseur, arrose];
arrose = _ref[0]
arroseur = _ref[1];
```

C'est bien plus verbeux et désagréable à lire et écrire. CoffeeScript dispose de nombreuses autres fonctionnalités permettant au programmeur de se simplifier la vie. Parmi ces fonctionnalités, citons :

- une syntaxe sans accolades inspirée de Python,
- les listes définies par compréhension, existant notamment en Python, Haskell ou Erlang,
- une notation courte pour les fonctions anonymes, omniprésentes dans les langages fonctionnels mais existants aussi dans des langages tels que C#, PHP ou Scala,
- ou encore la possibilité d'utiliser un système de classes et d'héritage plus classique.

CoffeeScript apporte un bénéfice tangible au développement JavaScript et fait partie des langages qui inspirent le comité ECMA Script. D'autres compilateurs vers JavaScript existent pour répondre à différents besoins.



Les *minifieurs* ont pour rôle de rendre un programme JavaScript le plus petit possible afin d'accélérer le transfert vers le navigateur. En minifiant et compressant jQuery, la taille du code passe de 247KB à 32KB : c'est un gain d'espace non négligeable ! Les *minifieurs* utilisent de [nombreuses techniques](#) pour gagner de l'espace. Ce tableau résume quelques-unes des opérations réalisées par UglifyJS, l'outil utilisé par jQuery :

Description	Avant	Après
Suppression des espaces	<code>var message = "Bonjour !"</code>	<code>var message="Bonjour !"</code>
Renommage	<code>var message = "Bonjour !"</code>	<code>var a = "Bonjour !"</code>
Conditions	<code>if (foo) return a(); else return b();</code>	<code>return foo?a():b();</code>
Indexation	<code>foo["bar"]</code>	<code>foo.bar</code>
Code "mort"	<code>var a = 1; if (a === 1) { ..... }</code>	<code>var a = 1;</code>

En octobre 2011, un autre langage a beaucoup fait parler de lui : Google Dart. Il est possible de le compiler vers JavaScript, mais l'objectif annoncé est de remplacer JavaScript entièrement. Dart a été influencé par Java, un langage largement répandu et pour lequel les approches possibles pour créer de [larges applications](#) sont bien comprises. Google Chrome devrait être le premier navigateur à supporter Google Dart nativement, au même rang que JavaScript. Parmi les [différents reproches](#) faits à Dart, notons le [système de types non sûr](#) et la [balkanisation du web](#). En effet, le risque de voir fleurir les sites web "optimisés pour Dart" et "optimisés pour Webkit" est important, ce qui est un danger pour le [web ouvert](#).



# DART

Logo Dart

CoffeeScript, UglifyJS et Dart ne sont que trois exemples de compilation vers JavaScript. Le nombre de [compilateurs capables de produire du JavaScript](#) est bien plus important, et il y a de fortes chances pour que votre langage favori soit dans la liste.

## Modules JavaScript

Malgré l'existence de nombreuses bibliothèques écrites en JavaScript, il n'y a pas de moyen **standard** de les écrire. Premièrement, il est difficile de cacher dans une bibliothèque les fonctions internes, que l'utilisateur ne doit pas manipuler. Ensuite, la gestion des dépendances se fait de manière totalement *ad hoc* : on inclut dans l'ordre les bibliothèques nécessaires nous-mêmes, au lieu de dire pour chaque bibliothèque quelles sont les dépendances, ce qui permettrait au moteur JavaScript de gérer le chargement lui-même.

Pour rendre notre propos plus concret, admettons qu'on veuille écrire un module JavaScript de correction orthographique. C'est une tâche potentiellement complexe nécessitant de découper notre code en de nombreuses fonctions, mais au final l'utilisateur ne va se servir que d'une seule fonction : `liste_erreurs()`, qui va retourner la liste des erreurs potentielles. Voyons un exemple d'utilisation d'une telle bibliothèque :

Code : HTML

```
<script type="text/javascript" src="correction.js"></script>

...

<script>
function surligner_erreurs(texte) {
    var erreurs = Correction.liste_erreurs(texte);
    for (var i in erreurs) {
        surligner(erreurs[i].debut, erreur[i].fin,
        erreurs[i].alternatives);
    }
}
</script>
```

On récupère ainsi la liste des erreurs potentielles, et la fonction `surligner()` (définie dans un autre module) s'occupe de



surligner le texte contenant les erreurs et d'ajouter un évènement qui se déclenchera lorsque l'utilisateur cliquera sur le texte surligné. L'utilisateur n'aura plus alors qu'à choisir la version correcte. Pour proposer une telle interface, la solution la plus simple est de placer dans `correction.js` un objet JavaScript :

#### Code : JavaScript

```
var Correction = {
  dictionnaire: undefined,

  // nécessaire de placer ces fonctions auxiliaires dans l'objet
  pour
  // ne pas polluer l'espace global
  init_dict: function(...) { ... },
  tokenize: function(...) { ... },
  contexte: function(...) { ... },
  erreurs_mot: function(...) { ... },

  liste_erreurs: function (texte) {
    var mots = tokenize(texte),
        erreurs = new Array();

    for(var mot in mots) {
      var contexte = contexte(mot, texte);
      if(this.erreur(mot, contexte)) {
        erreurs.push(erreurs_mot(mot, contexte));
      }
    }

    return erreurs;
  }
};

Correction.dictionnaire = Correction.init_dict();
```

Le problème qui se pose ici est que n'importe quel code est capable de modifier le dictionnaire ou d'appeler des fonctions internes de `Correction`. D'un point de vue conception, c'est problématique : toutes les variables et fonctions sont publiques. On est obligé de placer nos fonctions comme membres de l'objet, ce qui n'est pas agréable et moins pratique pour déboguer.

L'idée est donc de *cacher* `dictionnaire`, mais d'exposer les fonctions le manipulant. Comment cacher une variable ? En JavaScript, la portée des variables est la fonction. Encapsuler notre code dans une fonction nous permettra de n'exposer que les fonctions qui nous intéressent :

#### Code : JavaScript

```
var Correction = (function() {
  var dictionnaire;

  function init_dict(...) { ... },
  function tokenize(...) { ... },
  function contexte(...) { ... },
  function erreurs_mot(...) { ... },
  function liste_erreurs(...) { ... },

  dictionnaire = init_dict();

  return {
    liste_erreurs: liste_erreurs
  };
})();
```

Intéressons-nous ici d'abord à la déclaration de `Correction` (`var Correction = (function() { ... })()`). Pourquoi autant de parenthèses ? Il s'agit en fait ici de déclarer une fonction, de l'appeler immédiatement après sa déclaration,

puis de stocker l'objet reçu dans `Correction`. Notre variable `Correction` contient désormais `liste_erreurs` qu'il est possible d'appeler depuis l'extérieur. Grâce au concept de *closure*, `liste_erreurs` peut manipuler le dictionnaire, et c'est le seul moyen de le manipuler ! Nous venons ici de faire quelque chose de très intéressant : on n'expose à l'utilisateur que l'interface qu'il doit utiliser, et le reste est caché.

Des variantes de cette approche sont utilisées par de nombreuses bibliothèques, dont `jQuery` (cf. le [code](#)) qui commence avant tout par déclarer une fonction puis déclare à la fin du code que `window.jQuery = window.$ = jQuery;`, ce qui permet d'ajouter `jQuery` et `$` aux variables utilisables partout dans la page web.

## RequireJs



Logo Require.js

Comment déclarer les dépendances entre les différentes bibliothèques qu'on est amenés à utiliser dans le navigateur ? Il est naturellement possible d'inclure les fichiers les uns à la suite des autres, mais c'est une méthode primitive pour laquelle il est facile de se tromper. Un projet nommé [RequireJS](#) nous propose de spécifier les dépendances entre les différentes bibliothèques que nous utilisons, pour les charger dynamiquement ou créer un fichier JavaScript compact contenant uniquement le code nécessaire pour un site web donné.

Pour demander le chargement d'un fichier puis d'exécuter du code, il suffit d'utiliser `require()`. Ainsi, `require(["correction.js"], liste_erreurs);` se chargera d'appeler la fonction `liste_erreurs` une fois le chargement de correction effectuée.

Pour définir un module et ses dépendances, c'est tout aussi simple : il suffit d'utiliser `define`. Supposons que nous avons séparé la correction et la tokenization (qui est utilisée ailleurs). On peut demander le chargement de la tokenization de cette manière :

### Code : JavaScript

```
define(["./tokenize"], function() {  
    ...  
    return {  
        liste_erreurs: liste_erreurs  
    };  
});
```

RequireJS permet avant tout de faciliter le développement en permettant au développeur de découper ses fichiers et de déclarer ses dépendances comme il le souhaite. Il devient donc facile d'inclure des bibliothèques et leurs dépendances via un simple appel à `require()`, ce qui facilite le développement. Une fois que le site passe en production, il suffit de demander à RequireJS d'optimiser le code utilisé : regrouper les modules utilisés, et *minifier* le tout. Le développement web devient alors plus facile !

Un autre intérêt est qu'un code proprement séparé en modules offre de nouvelles opportunités d'optimisation. Au lieu d'envoyer tout le code JavaScript d'un coup, il devient possible d'identifier les modules nécessaires dès le chargement et ceux qui [peuvent au contraire attendre plus longtemps](#). C'est exactement ce qu'a décidé de faire Twitter en mai 2012, et cela a aidé à [réduire le temps d'affichage du premier tweet par cinq](#), ce qui n'aurait pas été possible sans un découpage propre en modules.

## Pourquoi se limiter au navigateur ?

Jusqu'ici, nous nous sommes intéressés à l'utilisation de JavaScript exclusivement dans le navigateur, ce qui est le rôle classique du langage. En effet, on utilise plutôt dans le [serveur](#) des langages tels que Python, Ruby ou encore PHP. C'est notamment le modèle qu'a adopté le Site du Zéro, en écrivant le serveur en PHP avec Symfony 2. C'est parfaitement raisonnable pour la plupart des applications, mais il s'avère que pour des applications très gourmandes en JavaScript, cela oblige souvent à écrire le même code deux fois : une fois en PHP et une autre fois en JavaScript.

Par exemple, pour un formulaire, on peut vouloir le valider en JavaScript avant de permettre l'envoi, ce qui laisse à l'utilisateur

l'opportunité de corriger ses erreurs au plus tôt. Un autre exemple est celui d'un panier web : le calcul du prix est souvent fait en JavaScript (pour une actualisation instantanée du prix) mais aussi sur le serveur lors de la validation finale. D'une manière générale, plus il y a de données à afficher et modifier dans une page web, plus le code sera **dupliqué sur le navigateur et le serveur** : une fois pour rendre la visite plus rapide et agréable, et l'autre fois pour traiter les données réelles et les sauvegarder plus tard.

Une solution pour éviter ce problème est d'écrire tout dans un langage donné, puis de convertir en JavaScript les parties qui vont s'exécuter sur le serveur web. C'est la solution adoptée par [OPA](#) et [Ocsigen](#), pour ne citer qu'eux.

Une autre solution qui semble avoir plus de popularité est d'utiliser JavaScript sur le serveur web ! Il n'y a pas alors à se demander comment est-ce que notre JavaScript va être généré. [Node.js](#) utilise cette approche, et permet aux développeurs de spécifier le comportement du serveur directement en JavaScript, en tirant avantage de la facilité de déclarer des fonctions pour encourager un style de programmation à base de callbacks, ce qui évite d'être handicapé par les entrées/sorties bloquantes. Pour créer un serveur, il suffit d'écrire le code suivant :



#### Code : JavaScript

```
var http = require('http');

http.createServer(function(request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.write("Hello World");
  response.end();
}).listen(8888);
```

On passe ici à `http.createServer` une fonction qui sera appelée pour chaque requête. Node.js réutilise le concept de la programmation événementielle : une boucle se charge de lancer les fonctions prêtes à être exécutées, c'est-à-dire n'attendant plus la fin d'une entrée/sortie telle que la lecture d'un fichier ou une requête vers une base de données. Typiquement, le code ci-dessus ne fait rien tant que personne n'accède au serveur, et exécute notre fonction une fois que quelqu'un y accède avec son navigateur.

Jusque-là, la seule différence avec le fonctionnement d'un serveur classique est l'utilisation de JavaScript. La vraie différence se fait sentir au moment de faire des entrées/sorties pendant le traitement d'une requête. C'est très fréquent : il faut bien accéder à une base de données quelconque pour savoir quelles informations afficher.

#### Code : JavaScript

```
var mysql = require("db-mysql");

new mysql.Database().connect(function() {
  this.query().select(["auteur",
    "texte"]).from("billets").limit(10)
    .execute(function(error, billets) {
      http.write("Affichage de " + length(billets) + "
billets");
      // Boucle d'affichage
    });
});
```

Dans ce code, on commence par se connecter à une base de données, puis on déclare une requête SQL. Pour schématiser, l'appel à `execute()` se fait en trois temps :

1. exécution de la requête SQL,
2. attente de la réponse,
3. et traitement du résultat (pour affichage, par exemple).

Node.js s'assure que le serveur ne chôme pas pendant la phase d'attente. Une fois que la requête est terminée, la boucle

d'événements s'assurera que la fonction passée à `execute` soit exécutée. C'est pour cette raison qu'on l'appelle *callback* (rappel) : elle est appelée une fois qu'on a besoin d'elle.

Alors que les entrées/sorties sont le problème numéro un des sites voulant passer à l'échelle, Node.js propose donc une alternative permettant de limiter ce problème. Bien sûr, ce n'est qu'un compromis, et d'autres problèmes spécifiques à Node.js se posent : par exemple, l'utilisation de nombreux callbacks imbriqués rend le code [plus difficile à lire](#). Des modules Node.js existent pour pallier ce problème, tels que [node-promise](#) où il suffit de spécifier les dépendances entre les fonctions, la bibliothèque se chargeant du reste.

## Le retour des modules

Nous venons de citer un module Node.js, et il existe en réalité des milliers. Le problème qui se posait dans le navigateur existe aussi sur le serveur, et Node.js a choisi de suivre une convention permettant aux modules de déclarer leurs noms et leurs dépendances. Concrètement, ceci se fait grâce au fichier `package.json` présent dans tout module Node.js. Par exemple, voilà comment nous pourrions déclarer notre module de correction orthographique :

### Code : JavaScript

```
{
  "name": "correction",
  "description": "Liste les alternatives possibles des mots mal orthographiés d'un texte",
  "version": "0.1.alpha1",
  "author": "Robert Larousse",
  "dependencies": {
    "tokenize": "0.3",
    "debug": "*"
  },
}
```

Ainsi, avant d'installer ce module, Node.js devra s'assurer d'avoir installé la version 0.3 de `tokenize` et une version quelconque de `debug`. Grâce à cette convention, une plateforme comme Node.js jouit d'un [large écosystème de modules](#) permettant à chaque développeur de ne pas avoir à réinventer la roue à chaque projet.



Logo CommonJS

Néanmoins, il existe d'autres plateformes exécutant du JavaScript : cela peut aller de la base de données CouchDB (où les « requêtes » sont écrites en JavaScript) aux applications utilisant JavaScript comme un langage de script. C'est le cas par exemple du jeu de stratégie [0 A.D.](#) qui utilise JavaScript pour [l'intelligence artificielle des unités](#). Est-il possible d'utiliser des modules

Node.js sur ces plateformes ?

À la manière de RequireJS dans le navigateur, la solution se nomme ici CommonJS. CommonJS a pour but de spécifier différentes conventions que peuvent suivre les bibliothèques écrites en JavaScript afin de coopérer entre elles. La spécification CommonJS la plus importante se nomme `Modules`, et spécifie comment un module JavaScript peut expliciter son nom, sa version, et les dépendances nécessaires. Les dépendances sont ensuite chargées automatiquement avec la fonction `require()`. C'est exactement ce qu'il faut faire avec Node.js, et pour cause : Node.js implémente la spécification `Modules`, ce qui permet à de nombreux modules d'être réutilisés par ailleurs.

## Un langage moderne

Nous l'avons vu : le langage JavaScript rend très difficile l'utilisation de certains concepts aussi simples que les modules, longtemps acquis dans d'autres langages. On pourrait penser que le langage n'a eu du succès que parce que c'était la seule solution dans le navigateur, mais ce serait oublier que c'est un langage qui reste moderne en offrant des fonctionnalités permettant d'écrire du code expressif et puissant et qui ne se trouvent pas nécessairement dans d'autres langages.

Les fonctions, tableaux ou encore les expressions régulières sont tous des objets qu'il est possible de manipuler, passer en paramètre et de retourner à la fin d'une fonction. Cela permet à JavaScript d'utiliser le paradigme fonctionnel quand c'est le plus adapté.

On compte aussi parmi les fonctionnalités le tout aussi controversé que puissant "prototype". C'est en effet une fonctionnalité relativement peu utilisée par les développeurs permettant entre autres d'obtenir de l'héritage. Intéressons-nous ici plutôt à l'ajout de fonctionnalité à des objets existants. Par exemple, si vous désirez tester l'égalité de deux `Array`, il n'y a pas de moyen simple de faire ça. C'est donc l'opportunité de déclarer une telle fonction :

**Code : JavaScript**

```
Array.prototype.equals = function (otherArray) {  
    if (!Array.isArray(otherArray)) {  
        return false;  
    }  
  
    return this.every(function (element, i) {  
        if (Array.isArray(element)) {  
            return element.equals(otherArray[i]);  
        } else {  
            return element === otherArray[i];  
        }  
    });  
};
```

Désormais, `[1, 2, 3].equals([1, 2, 4])` renvoie faux. Cette astuce est utilisée pour s'assurer que les fonctions ajoutées dans ECMAScript 5 seront disponibles dans tous les navigateurs : si elles n'existent pas, il suffit de les créer. On appelle ceci la "feature detection". Elle permet aux développeurs d'écrire du JavaScript moderne qui sera exécuté rapidement dans les navigateurs récents, mais qui fonctionnera tout de même dans les navigateurs les plus vieux. C'est ainsi que [es5-shim](#) émule de nombreuses fonctions ajoutées dans ECMAScript 5 et que [Modernizr](#) apporte des fonctionnalités ajoutées récemment dans HTML5 et CSS3. À chaque fois, les navigateurs récents utilisent une implémentation efficace, et les autres exécuteront une émulation de ces fonctionnalités : moins rapide, mais fonctionnelle. L'adoption des technologies récentes n'est donc pas freinée par les vieux navigateurs.

Une autre force de JavaScript se situe dans les *closures*, ou fermetures en français. En pratique, ce sont simplement des fonctions qui sont déclarées dans le corps d'une autre fonction, et qui accèdent aux membres de la fonction parente. C'est un concept à la fois simple et puissant qui permet par exemple de [simuler les membres privés](#) en JavaScript, comme nous l'avons vu plus haut dans notre module [Correction](#). Une des possibilités offertes par les closures est l'amélioration des performances en ne calculant qu'une fois certaines expressions. C'est avantageux pour les accès au DOM qui peuvent être long suivant le navigateur. Voyez ce code :

**Code : JavaScript**

```
//exemple sans closure  
var addDescription = function (texteDescription) {  
    var baliseDescription = document.getElementById('description');  
    //appel au DOM  
    var child = document.createElement('p');  
    child.innerHTML = texteDescription;  
    baliseDescription.appendChild(child);  
}  
addDescription('première description'); //un appel au DOM  
addDescription('deuxième description'); //deuxième appel au DOM ...  
  
//avec closure  
var addDescription = (function () {  
    var baliseDescription = document.getElementById('description');  
    //appel au DOM  
    return function (texteDescription) {  
        var child = document.createElement('p');  
        child.innerHTML = texteDescription;  
        baliseDescription.appendChild(child);  
    }  
})(); //premier appel au DOM  
  
addDescription('première description'); //pas d'appel au DOM  
addDescription('deuxième description'); //pas d'appel au DOM ...
```

Pour vous assurer du gain de performance entre différentes solutions que vous avez codé, vous pouvez vous rendre sur le site JSPerf. [Un petit test pour l'exemple ci-dessus](#) permet par exemple de comparer les performances des deux versions suivant les navigateurs.

## ECMAScript Harmony

Comme nous le disions dans l'historique, le comité TC39 responsable de l'évolution d'ECMAScript travaille actuellement sur la prochaine version : ECMAScript Harmony. Afin d'éviter les déboires de la version 4 qui n'a jamais abouti et de prendre en compte les différents usages de JavaScript, des objectifs précis ont été **définis**. L'objectif numéro un est de concevoir un langage plus adapté pour :

- les applications complexes,
- les bibliothèques utilisées par ces applications,
- et les compilateurs vers JavaScript.

Il ne s'agit donc pas de créer un langage qui soit « théoriquement mieux », mais bien de répondre à des problèmes **concrets**. Le deuxième objectif est de produire une spécification qui puisse être implémentée en JavaScript : cela permettra de :

- utiliser et de tester ECMAScript 6 avant qu'il soit prêt dans nos navigateurs,
- d'avoir à disposition une implémentation de référence,
- et potentiellement d'utiliser cet interpréteur sur les navigateurs les plus réticents, à la manière d'[es5-shim](#) pour ECMAScript 5.

Parmi les objectifs restants, un troisième objectif est d'intégrer directement dans le langage les patrons de conception les plus [répandus en JavaScript](#) pour les intégrer dans le langage. Paul Graham de YCombinator se demande dans un de [ses essais](#) si les patrons de conceptions ne sont pas l'œuvre d'un "compilateur humain". Si c'est le cas, c'est certainement une indication d'un manque dans le langage qui ne demande qu'à être comblé. En JavaScript, le manque le plus évident en concerne les modules.

## Modules

Comme nous l'avons vu précédemment, les "modules" ont été développés sans l'aide réelle du langage, mais seulement avec des closures (IIFE) et des conventions (CommonJS). ECMAScript Harmony souhaite reprendre ces excellentes idées en les intégrant dans le langage pour imposer ces standards établis, et les rendre plus facile à utiliser. On peut ainsi voir les modules en ECMAScript Harmony comme du sucre syntaxique : on pourrait faire sans, mais ils sont désormais bien plus faciles à écrire. Par conséquent, la bibliothèque décrite plus haut peut se réécrire plus simplement :

### Code : JavaScript

```
import "tokenize.js";

module Correction {
  var dictionnaire = init_dict();

  function init_dict(...) { ... }
  ...

  export function liste_erreurs() { ... }
}
```

Il sera désormais bien plus simple d'écrire des modules sans erreur, en spécifiant facilement les méthodes et variables qu'on veut rendre disponibles à tout le monde et les dépendances nécessaires, à la façon de CommonJS et RequireJS. Mieux, cette nouvelle syntaxe permet d'expliquer qu'on est bien en train d'écrire un module destiné à être utilisé ailleurs, ce qui améliore la lisibilité du code. Un langage comme CoffeeScript pourrait aisément implémenter une telle syntaxe, étant donné qu'elle est de toute façon destinée à être employée dans le futur.

## Templates

Dès lors qu'on fait un appel AJAX en JavaScript, il est nécessaire de présenter à l'utilisateur les données reçues. Ces données

sont souvent sous la forme de JSON, et il faut donc les transformer en HTML avant affichage. C'est une tâche très classique en développement web et il existe des dizaines de [moteurs de template](#) sur le web. Sur le Site du Zéro, karamilo a même développé les [KaraTemplates](#) qui ont été utilisés pour faciliter l'écriture de contenu sur le site. PHP lui-même a une syntaxe pour écrire des templates facilement : par exemple, `<?= $score ?>` est équivalent à `<?php echo $score; ?>`.

En JavaScript, la situation est toute autre, et il n'y a pas actuellement de solution simple et standard pour écrire des templates. Prenons l'exemple de jQuery : les [jQuery templates](#) ont été dépréciées et ne sont plus maintenues, et la solution du futur, [JsRender](#) n'est même pas encore en beta. D'autres solutions populaires tels que [Mustache](#) ne proposent pas de protection et ne garantissent pas que le HTML rendu sera bien formé. Cela témoigne à la fois de la difficulté de l'écriture d'une telle solution en JavaScript pur et du besoin de tels templates. C'était l'occasion pour le comité de proposer une solution pour écrire des templates qui soit facile à utiliser, sécurisée et performante : les [quasi-literals](#). Voici un exemple :

#### Code : JavaScript

```
url = "http://example.com/",
message = query = "Hello & Goodbye",
color = "red",
safehtml`<a href="${url}?q=${query}"
onclick=alert (${message})>${message}</a>`
```

(Vous remarquerez que la coloration syntaxique n'est pas encore adaptée à ECMAScript Harmony)

Cette syntaxe n'est en fait que du [sucre syntaxique](#). Plus précisément, le code écrit plus haut est strictement équivalent au charabia suivant :

#### Code : JavaScript

```
var $$callSite0 = Object.freeze({
  raw: Object.freeze(["<a href=\"", "?q=\"", "\"
onclick=alert(\"\", \">\", \"</a>\""]),
  cooked: Object.freeze(["<a href=\"", "?q=\"", "\"
onclick=alert(\"\", \">\", \"</a>\""])
});
url = "http://example.com/",
message = query = "Hello & Goodbye",
color = "red",
safehtml($$callSite0, (url), (query), (message), (color), (message))
```

La méthode `safehtml` sera intégrée à JavaScript, et lors de son appel, elle produit simplement le résultat HTML suivant :

#### Code : HTML

```
<a href="http://example.com/?q=Hello%20%26%20Goodbye"
onclick=alert (&#39;Hello&#32;\x26&#32;Goodbye&#39;)
style="color: red">
Hello &amp; Goodbye
</a>
```

Les sauts à la ligne ont été insérés manuellement, mais l'idée est là : `safehtml` vous permet de produire du HTML sécurisé en toute confiance étant donné que la fonction prend en compte le contexte et sait quels caractères il faut transformer. On évite non seulement le HTML mal formaté qui peut "casser" l'affichage d'un site web mais aussi des surprises telles que les [failles XSS](#) qui sont une plaie réelle pour de nombreux sites web, y compris le Site du Zéro qui en a corrigé plusieurs au fil des années et des petits rigolos. Il est naturellement possible de définir sa propre fonction qui pourra avoir d'autres intérêts : par exemple l'internationalisation, en choisissant le texte à afficher suivant la langue de l'utilisateur.

Plus généralement, c'est aussi un très bon moyen d'écrire un [Domain-Specific Language \(DSL\)](#). Voyons quelques exemples simples :

Domaine	Code
appel jQuery	<code>\$`a.\${className}[href=~'//\${domain}/']`</code>



Requête HTTP	GET`http://example.org/service?a=\${a}&b=\${b}`
Expression rationnelle	re`\d+(\${localeSpecificDecimalPoint}\d+)?`
Chaîne brute	raw`In JavaScript '\n' is a line-feed.`

Les templates sont donc un bon exemple des objectifs que s'est fixé ECMAScript Harmony. Cette nouvelle fonctionnalité :

- correspond à un problème récurrent en JavaScript ;
- dispose d'une implémentation pouvant être testée dès aujourd'hui ;
- s'intègre bien dans le langage et offre de nombreuses possibilités (cf. tableau plus haut) ;
- s'inspire de bonnes idées existantes dans d'autres langages : (les [macros hygiéniques de Scheme](#), l'[inférence de type](#), ou encore les [quasi-quotes de Lisp](#)).

## Quelques autres nouveautés

Ces deux nouveautés ne sont pas les seules prévues ! Parmi les nouveautés présentées sur le wiki ECMAScript, nous en citerons trois. Les **destructuring assignments** proposés par CoffeeScript seront intégrés au langage et auront un comportement uniforme dans de nombreux cas :

### Code : JavaScript

```
// échanger deux variables
[a, b] = [b, a];

// retourner plusieurs valeurs dans une fonction
var [tete, queue] = decouperListe(l);

// déclarer une fonction prenant en paramètre un objet
function showPixel({"x": x, "y": x}) { ... }

// boucler sur une liste d'objet
for (let [clef, valeur] in obj) { ... }
```

La **notation courte pour les lambdas** permet d'écrire des fonctions anonymes plus simplement. Voyons la différence en reprenant un exemple déjà utilisé :

### Code : JavaScript

```
var ages = [12, 89, 23, 45, 5, 37];
var majeurs = ages.filter(function(age) { return age > 18; });
var majeurs = ages.filter(age => age > 18)
// majeurs == [89, 23, 45, 37];
```

Cette fois, ce n'est pas uniquement la syntaxe qui change, et il y a une légère différence quant à la valeur de [this](#) dans chacune des fonctions. Néanmoins, dans tous les cas où **this** n'est pas utilisé, ces deux formes sont équivalentes.

Dans cette nouvelle version d'ECMAScript, les compilateurs vers JavaScript ne seront pas en reste. En particulier, une nouvelle forme de tableaux va faire son apparition. En effet, les tableaux JavaScript ne sont que des objets : l'indice est stocké sous forme de chaîne ce qui implique une [conversion entier/chaîne](#) à l'exécution. Ainsi, le tableau `[23, 42, 118]` est en fait équivalent à l'objet `{"0": 23, "1": 42, "2": 118, "length": 3}`, ce qui est une source importante d'inefficacité. C'est pour cette raison qu'ECMAScript Harmony facilitera la manipulation de [données compactes](#) : il incorpore des tableaux typés ([inspirés de WebGL](#)) et des *bit fields* qui permettront de représenter des données binaires de manière compacte et efficace. C'est une fonctionnalité essentielle pour les langages compilant vers JavaScript. Ainsi, [LLJS](#) utilise ces tableaux pour compiler un sous-ensemble du JavaScript vers... du JavaScript. La différence est que la mémoire est gérée explicitement via les tableaux typés, ce qui rend le code illisible pour un humain mais qui nécessite beaucoup moins d'efforts de la part du [ramasse-miettes](#) du moteur






JavaScript exécutant le code, et permettra potentiellement de rendre le code plus lisible.

Il est enfin intéressant de noter que `let` pourra être utilisé comme `var` mais avec une portée limitée au bloc au lieu de la fonction, ce qui évitera nombre de confusions : la portée des variables est actuellement celle de la fonction entière, ce qui n'est pas le cas dans la plupart des langages de programmation.

Pour terminer, sachez qu'il est possible de participer au développement d'ECMAScript Harmony en tant que simple programmeur JavaScript. Partant du constat que faire partie du comité n'était pas accessible à moins d'être embauché par une association ou une université, des développeurs ont décidé de se réunir pour faire remonter au comité tous les problèmes qu'ils ont, à la fois avec la version actuelle de JavaScript et avec les propositions de la version suivante. Cette initiative s'appelle JSFixed et un bon nombre de points a déjà été identifié. Si votre niveau d'expertise en anglais et en JavaScript vous permet de participer à la discussion, c'est l'occasion parfaite de participer à l'avenir de JavaScript.

JavaScript ne se limite donc pas à faire clignoter votre page web, loin s'en faut. Les possibilités sont sans cesse plus nombreuses : il est entièrement réaliste aujourd'hui de n'utiliser que JavaScript pour un site web. Sa popularité est grandissante, et ses évolutions sont prometteuses : c'est donc un langage à ne pas négliger qui est là pour rester. Il serait dommage de l'ignorer.

## Sources

-  YUI Theater — Dave Herman: “The Future of JavaScript” (48 min.)
-  JavaScript: The World's Most Misunderstood Programming Language
-  ECMAScript Wiki (modules, quasis, destructuring, iterators, let, shorter function syntax)
-  Slides de la présentation de Node.js à la jsconf 2009
-  JavaScript Weekly

Merci à Hellish, Arthur, Nesquik69, bluestorm, Golmote et Kineolyan pour leurs relectures et à l'équipe du site pour son travail de l'ombre.

## Partager

