



A BOOK APART

Les livres de ceux qui font le web

No.

20

Mat Marquis

JAVASCRIPT POUR LES WEB DESIGNERS

PRÉFACE DE Lara Hogan

EYROLLES



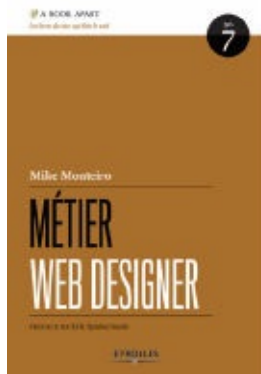
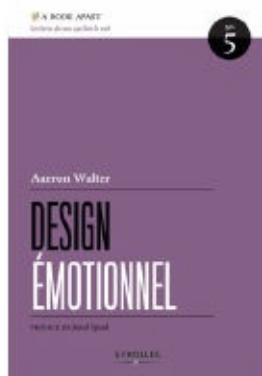
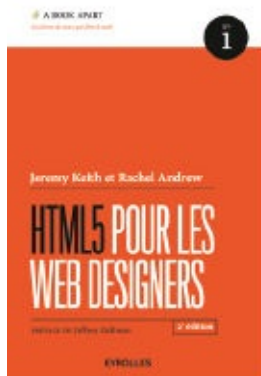
Résumé

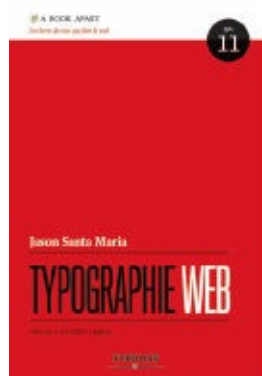
Au fil de nombreux exemples pratiques, initiez-vous aux règles de syntaxe, aux fondamentaux des scripts, ainsi qu'à la gestion des types de données et des boucles. Nul doute que ce tour d'horizon vous donnera confiance ; vous aurez une vision plus claire de JavaScript et serez fin prêt à vous mettre au travail !

Au sommaire Mise en route * Inclure JavaScript dans une page * Nos outils de développement * Les règles fondamentales * Nous voilà prêts * **Comprendre les types de données** * Types primitifs * Types d'objets * **Les expressions conditionnelles** * Instructions if/else * Opérateurs de comparaison * Opérateurs logiques * switch * J'ai le tournis * **Les boucles** * for * while * continue et break * Boucles infinies * Et maintenant, tous ensemble * **Scripter avec le DOM** * window * Utiliser le DOM * Scripter avec le DOM * Événements DOM * Amélioration progressive

Biographie auteur

Vous avez des sueurs froides dès que vous entendez parler de JavaScript ? Respirez un grand coup et prenez votre courage à deux mains : **Mat Marquis** est ici à vos côtés pour vous offrir une visite rapide mais efficace de ce langage aujourd'hui incontournable.





www.editions-eyrolles.com

Mat Marquis

JAVASCRIPT POUR LES WEB DESIGNERS

EYROLLES

The logo for Eyrolles, featuring the word "EYROLLES" in a bold, sans-serif font. Below the text is a horizontal line with a small red dot in the center.

ÉDITIONS EYROLLES

61, bld Saint-Germain

75240 Paris Cedex 05

www.editions-eyrolles.com

Traduction autorisée de l'ouvrage en langue anglaise intitulé *JavaScript for Web Designers* de Mat Marquis (ISBN : 978-1-937557-47-8), publié par A Book Apart LLC

Adapté de l'anglais par Charles Robert

© 2016 Mat Marquis pour l'édition en langue anglaise

© Groupe Eyrolles, 2017, pour la présente édition, ISBN : 978-2-212-67408-8

Dans la même collection

HTML5 pour les web designers - n°1, 2^e édition, Jeremy Keith et Rachel Andrew, 2016.

CSS3 pour les web designers - n°2, 2^e édition, Dan Cederholm, 2016.

Stratégie de contenu web - n°3, Erin Kissane, 2011.

Responsive web design - n°4, 2^e édition, Ethan Marcotte, 2017.

Design émotionnel - n°5, Aarron Walter, 2012.

Mobile first - n°6, Luke Wroblewski, 2012.

Métier web designer - n°7, Mike Monteiro, 2012.

Stratégie de contenu mobile - n°8, Karen McGrane, 2013.

La phase de recherche en web design - n°9, Erika Hall, 2015.

Sass pour les web designers - n°10, Dan Cederholm, 2015.

Typographie web - n°11, Jason Santa Maria, 2015.

Web designer cherche client idéal - n°12, Mike Monteiro, 2015.

Design web responsive et responsable - n°13, Scott Jehl, 2015.

Design tactile - n°14, Josh Clark, 2016.

Responsive design patterns - n°15, Ethan Marcotte, 2016.

Git par la pratique - n°17, David Demaree, à paraître.

Attention : la version originale de cet ebook est en couleur, lire ce livre numérique sur un support de lecture noir et blanc peut en réduire la pertinence et la compréhension.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands Augustins, 75006 Paris.

TABLE DES MATIÈRES

Préface

Introduction

CHAPITRE 1

Mise en route

CHAPITRE 2

Comprendre les types de données

CHAPITRE 3

Les expressions conditionnelles

CHAPITRE 4

Les boucles

CHAPITRE 5

Scripter avec le DOM

Conclusion

Ressources

Remerciements

Index

PRÉFACE

COMME BEAUCOUP DE GENS dans la communauté du développement front-end, à mesure que j'ai appris et développé mes compétences, j'ai aussi été amenée à me spécialiser dans divers aspects de la discipline. J'ai acquis une solide expertise du HTML, de CSS et de tout ce qui permet de rendre un site web plus performant. Mais j'ai eu beau faire des progrès, la liste des outils et des techniques à découvrir ne cesse de croître et de m'intimider.

JavaScript est lui aussi sur cette liste, et pendant bien des années, j'ai réussi à éviter de trop en apprendre à son sujet. Au lieu de ça, je comptais sur Google pour comprendre les messages d'erreur, je copiais-collais des scripts pour essayer de les bidouiller, sans véritablement chercher à comprendre le code.

Dix ans après le début de ma carrière, je me suis retrouvée avec une copie de *JavaScript pour les web designers* entre les mains, et j'ai ressenti ce même pincement au cœur. JavaScript était encore une langue étrangère pour moi ; jusqu'alors, j'avais pu compter sur les moteurs de recherche et la bonté de parfaits inconnus pour me dépanner. Qu'est-ce qui m'empêchait de m'attaquer enfin à ce monstre de technologie ?

Dix pages plus tard, ma crainte de JavaScript, cette bête nébuleuse, s'était envolée.

J'en ai appris plus sur JavaScript dans ce petit livre qu'au cours de toute ma carrière. Avec le style de Mat, JavaScript est incroyablement facile à apprendre. Des types de données aux boucles, je suis parvenue à pénétrer ce langage que je croyais insondable et à en tirer des leçons. Mat commence par aborder les concepts fondamentaux et ne précipite pas les choses, guidant sereinement le lecteur au fil d'exemples faciles à mettre en œuvre.

Mais ce livre n'est pas seulement utile pour apprendre JavaScript ; son superpouvoir, c'est de calmer aussi efficacement toutes vos craintes. J'ai refermé ce livre le cœur résolu : je peux enfin aborder et utiliser JavaScript sans avoir des sueurs froides devant ma console. Quelle que soit votre formation – designer, développeur, débutant complet ou juste novice en JavaScript – vous verrez que *JavaScript pour les web designers* est un livre accessible et divertissant. J'ai hâte que vous tourniez cette page et que vous vous plongiez dans sa lecture.

Lara Hogan

INTRODUCTION

EN TOUTE FRANCHISE, je devrais commencer ce livre par des excuses – pas à vous, lecteur, quoique je vous en devrai sans doute au moins une d’ici à la fin de ce livre. Non, c’est à JavaScript que je dois des excuses, pour toutes les ignominies que je lui ai dites au début de ma carrière – des injures si corrosives qu’elles feraient fondre du verre trempé.

C’est ma façon peu subtile de dire que JavaScript peut être un langage délicat à apprendre.

HTML et CSS présentent eux aussi une dose de difficulté, mais on peut les apprendre « sur le tas ». Ils sont plus simples dans le sens où, lorsqu’on tape quelque chose, cette chose se produit : `border-radius` arrondit les coins, la balise `p` ouvre un paragraphe. Rien de bien compliqué.

Quand je me suis mis à JavaScript, en revanche, tout ce que j’apprenais me paraissait être le sommet d’un nouvel iceberg terrifiant, un maillon de plus dans une chaîne de concepts interconnectés, chacun plus complexe que le précédent, des variables, une logique, un vocabulaire et des mathématiques que je croyais tout simplement hors de ma portée. Je tapais quelque chose, mais cela ne voulait pas toujours dire ce que je souhaitais exprimer. Si je collais un bout de code au mauvais endroit, JavaScript se mettait en colère. Lorsque je commettais une erreur, mon script tombait en panne.

Si vous avez l’impression que je viens de plagier les pages les plus angoissantes de votre journal intime et que vous avez déjà jeté ce livre à travers la pièce, rassurez-vous : si je dis tout ça, ce n’est pas pour raviver votre peur de JavaScript, mais pour vous dire que moi aussi j’avais peur. Il n’y a pas si longtemps encore, je copiaais-collais des scripts prémâchés de qualité douteuse, puis je croisais les doigts en rafraîchissant la page. Je pensais qu’un tel langage de programmation était impossible à maîtriser, et j’étais certain de n’être pas fait pour ça. J’étais un développeur, bien sûr, mais pas un développeur-développeur. Je n’avais pas le cerveau robotisé que ce travail exige. Je ne faisais que décorer des pages web pour gagner ma croûte.

J’étais intimidé.

Vous vous trouvez peut-être dans la même situation que moi à cette époque : vous vous tenez au bord d’un précipice de plus de cent pages, vous préparant à être complètement submergé par un torrent d’informations complexes dès la page dix. Je ne vous ferai pas ça, et je m’efforcerai de ne pas vous décourager.

JavaScript n’est pas la chose la plus simple à comprendre, et mon but avec ce livre n’est pas d’explorer le langage de fond en comble. Je ne connais pas tout

sur JavaScript, et je suis certain que personne ne peut s'en targuer. Mon objectif est de vous aider à atteindre le point le plus important dans l'apprentissage de n'importe quel langage : ce que l'on pourrait appeler le « déclic ».

C'est quoi au juste, JavaScript ?

Le nom lui-même peut prêter à confusion. Le mot « Java » évoque peut-être en vous des images d'« applets » Java sur de vieux navigateurs, ou un langage de programmation côté serveur. Lorsque JavaScript a fait sa première apparition, en 1995 (<http://bkaprt.com/jsfwd/00-01/>), il s'appelait « LiveScript », en référence au fait que le code s'exécutait dès qu'il était chargé par le navigateur. Mais en 1995, Java était très tendance, et les deux langages partageaient quelques similarités de syntaxe, alors pour une pure question de marketing, « LiveScript » est devenu « JavaScript ».

JavaScript est un langage de script léger, mais extrêmement puissant. Contrairement à bien d'autres langages de programmation, JavaScript n'a pas besoin d'être traduit en un format compréhensible par le navigateur : il n'y a pas d'étape de compilation. Nos scripts sont transmis plus ou moins en même temps que nos autres ressources (balisage, images et feuilles de styles) et sont interprétés à la volée.

C'est dans le navigateur qu'on le rencontre le plus souvent, mais JavaScript s'est infiltré partout, des applications natives aux livres électroniques. C'est l'un des langages de programmation les plus répandus au monde, et notamment parce qu'on le retrouve dans de nombreux environnements différents. JavaScript n'a pas besoin de grand-chose pour tourner : il demande seulement la présence d'un moteur JavaScript pour être interprété et exécuté et, tout comme certains navigateurs, les moteurs JavaScript sont open source. Des développeurs appliquent ces moteurs dans de nouveaux contextes, et c'est ainsi qu'on se retrouve avec des serveurs web (<http://nodejs.org>) et des robots artisanaux (<http://johnny.five.io>) programmés en JavaScript. Nous étudierons JavaScript dans le contexte du navigateur, mais si vous vous sentez l'âme d'un savant fou après avoir fini ce livre, j'ai une bonne nouvelle pour vous : la syntaxe que vous apprenez ici vous servira peut-être un jour à programmer votre Étoile noire.

A screenshot of a web form with a light blue sidebar on the left. The form has two input fields. The first field is labeled 'NOM D'UTILISATEUR' and contains the text 'MonsieurMuscle'. Below this field, a red error message reads 'Désolé, ce nom est déjà utilisé'. The second field is labeled 'MOT DE PASSE' and is currently empty.

FIG 0.1 : Valider le contenu d'un champ à mesure que les données sont saisies, plutôt qu'une fois le formulaire validé : un exemple classique d'amélioration employant JavaScript.

La couche interactive

JavaScript nous permet d'ajouter une couche d'interaction sur nos pages en complément de la couche structurelle, composée en HTML, et de la couche de présentation, notre feuille de styles CSS.

Il nous permet de contrôler les interactions de l'utilisateur avec la page dans les moindres détails ; ce contrôle s'étend même au-delà de la page elle-même et nous permet de modifier les comportements intégrés du navigateur. Prenons l'exemple simple d'un comportement de navigateur modifié par JavaScript que vous avez sûrement rencontré à de nombreuses reprises : la validation des champs d'un formulaire. Avant même que le formulaire ne soit envoyé, un script de validation parcourt tous les champs associés, vérifie leurs valeurs selon un ensemble de règles, puis autorise ou non l'envoi du formulaire ([FIG 0.1](#)).

Avec JavaScript, nous pouvons élaborer des expériences plus riches pour nos utilisateurs, des pages qui répondent à leurs interactions sans avoir besoin de les rediriger vers une nouvelle page, même lorsqu'ils doivent obtenir de nouvelles données auprès du serveur. Il nous permet également de combler les lacunes quand les fonctionnalités de base d'un navigateur ne suffisent pas, de corriger des bugs, ou encore de porter de nouvelles fonctionnalités sur les vieux navigateurs qui ne les prennent pas en charge de façon native. En bref, JavaScript nous permet de créer des interfaces plus sophistiquées qu'avec l'HTML et CSS seuls.

Ce que JavaScript n'est pas (ou plus)

Vu l'influence qu'il peut avoir sur le comportement du navigateur, on comprend facilement pourquoi JavaScript n'a pas toujours eu bonne réputation. Pour rendre une page inutilisable avec CSS, il faut vraiment y

mettre du sien. Une instruction comme `body { display: none; }` ne se retrouve pas dans une feuille de styles par hasard, quoique ça pourrait bien m'arriver un jour. Il est encore plus difficile de faire une erreur de balisage qui empêcherait complètement la page de fonctionner : une balise `strong` laissée ouverte par erreur donnera peut-être à la page un aspect étrange, mais elle n'empêchera personne de la consulter. Et lorsque la CSS ou le balisage sont la cause de problèmes importants, c'est généralement visible, de sorte que si le balisage HTML ou la feuille de styles CSS rendent la page impraticable, nous devrions le voir au cours de nos tests.

JavaScript, lui, est un drôle d'animal. Un exemple : vous ajoutez un petit script pour valider une adresse postale saisie dans un champ de formulaire, et la page est rendue comme prévu. Vous saisissez l'adresse « 20 rue de la Paix » pour tester le script et, en l'absence d'erreur, vous pensez que votre script de validation de formulaire fonctionne correctement. Mais si vous ne faites pas attention aux règles de votre script de validation, un utilisateur avec une adresse inhabituelle pourrait bien se trouver incapable de saisir des informations valides. Pour que le test soit complet, il faudrait essayer autant d'adresses bizarres que possible ; et vous en oublieriez sans doute quelques-unes.

Quand le Web était plus jeune et que la profession de développeur web en était à ses premiers balbutiements, nous n'avions pas de bonnes pratiques clairement définies pour gérer les améliorations en JavaScript. Il était pratiquement impossible d'employer des tests reproductibles, et la prise en charge des navigateurs était plus qu'intermittente. Cette combinaison a donné naissance à de nombreux scripts boiteux qui se sont échappés dans la nature. Dans le même temps, des individus parmi les moins recommandables du Web se sont soudainement retrouvés avec le pouvoir d'influer sur le comportement des navigateurs, refrénés uniquement par des limites qui étaient incohérentes au mieux, inexistantes au pire. Cela va de soi, ils n'ont pas toujours utilisé ce pouvoir à bon escient.

À cette époque, JavaScript était très critiqué. Il était jugé peu fiable, voire dangereux ; un moteur occulte tout juste bon à afficher des pop-ups.

Mais les temps ont changé. Les efforts de standardisation du Web, qui nous ont amené un balisage plus sémantique et une prise en charge satisfaisante de CSS, ont également rendu la syntaxe de JavaScript plus cohérente d'un navigateur à l'autre, et établi des contraintes raisonnables quant aux comportements du navigateur qu'il peut influencer. Parallèlement, des frameworks d'« assistance » JavaScript comme jQuery, développés sur la base de bonnes pratiques et conçus pour normaliser les excentricités et les bugs des navigateurs, aident aujourd'hui les développeurs à écrire des scripts

mieux conçus et plus performants.

Le DOM : comment JavaScript communique avec la page

JavaScript communique avec le contenu de nos pages par le biais d'une API appelée Document Object Model, ou DOM (<http://bkaprt.com/jsfwd/00-02/>). C'est le DOM qui nous permet d'accéder au contenu d'un document et de le manipuler avec JavaScript. De même que l'API de Flickr nous permet de lire et d'écrire des informations à travers son service, l'API DOM nous permet de lire, de modifier et de supprimer des informations dans des documents – de changer le contenu de la page web elle-même.

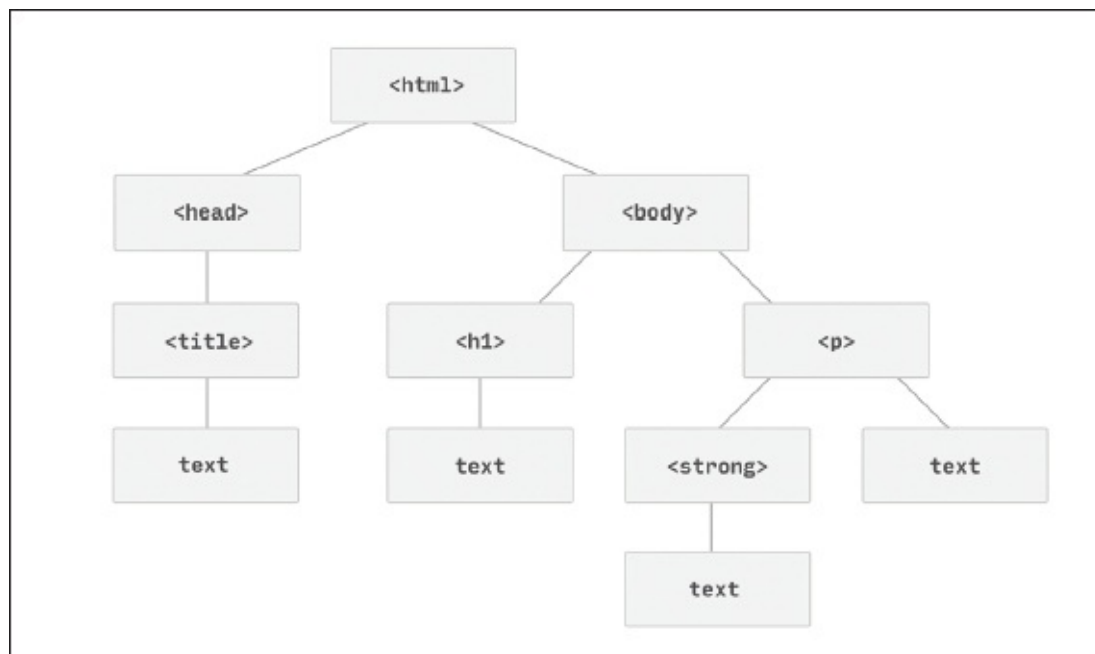


FIG 0.2 : L'expression « arbre DOM » prend tout son sens quand on fait le poirier.

Le DOM remplit deux fonctions. Il fournit à l'intention de JavaScript une « carte » structurée de la page en traduisant notre HTML dans un format que JavaScript (et de nombreux autres langages) peut comprendre. Sans le DOM, JavaScript serait incapable d'interagir avec le contenu du document. Chaque petit morceau de notre document est un « nœud » auquel JavaScript peut accéder *via* le DOM. Chaque élément, commentaire et petit bout de texte est un nœud.

La seconde fonction du DOM consiste à fournir à JavaScript un ensemble de méthodes et de fonctions permettant d'accéder aux nœuds cartographiés – de dresser une liste de toutes les balises `p` du document, par exemple, ou de

recueillir tous les éléments comportant une classe `.toggle` et un élément parent `.collapsible`. Ces méthodes sont standardisées dans tous les navigateurs, avec des noms accrocheurs comme `getElementsByTagName` ou `createTextNode`. Ces méthodes étant intégrées dans JavaScript, il peut être difficile de savoir où JavaScript prend fin et où le DOM débute, mais par chance, nous n'avons pas besoin de nous en préoccuper pour le moment.

C'est parti

Au fil de ce livre, nous apprendrons les règles de JavaScript par la pratique, nous barboterons gentiment dans le DOM et nous décortiquerons de vrais scripts pour voir ce qui les fait tourner. Mais avant de nous mesurer à cette bête qu'est JavaScript, nous allons tâcher de nous familiariser avec les outils de notre nouveau métier.

J'ai une formation de charpentier, et si quelqu'un m'avait envoyé sur un toit le premier jour, je ne suis pas sûr que j'y aurais survécu. Alors, dans le prochain chapitre, nous aborderons les outils de développement et les environnements de débogage intégrés des navigateurs modernes, et nous mettrons en place un environnement de développement pour commencer à écrire quelques scripts.

1

MISE EN ROUTE

AVANT DE NOUS AVENTURER sur le terrain de JavaScript, commençons par nous familiariser avec notre environnement de développement. Nous allons d'abord créer une page de travail, puis examiner cette page à travers le prisme des outils de débogage intégrés de nos navigateurs. Pour finir, nous aborderons les règles de base de la syntaxe JavaScript.

INCLURE DU JAVASCRIPT DANS UNE PAGE

Si vous avez déjà travaillé avec CSS, vous constaterez que l'inclusion d'un script dans votre page suit plus ou moins le même modèle que l'inclusion d'une feuille de styles, quoiqu'avec une syntaxe légèrement différente et quelques petites réserves.

Les scripts peuvent être inclus dans le document lui-même en les encadrant par des balises `<script></script>`, comme vous utiliseriez `<style></style>` pour inclure votre CSS.

```
<html>
  <head>
    ...
    <script>
      // Placez votre script ici.
    </script>
  </head>
```

On retrouve les mêmes inconvénients que pour les styles : si un même script est destiné à être utilisé sur plusieurs pages, il est contre-productif de le copier-coller dans chaque document. C'est un cauchemar à entretenir si vous voulez modifier votre script et appliquer le changement sur toutes les pages en même temps.

Heureusement, de même qu'on peut utiliser une même feuille de styles sur plusieurs pages, on peut facilement référencer un script externe. Là où CSS utilise la balise `link`, JavaScript emploie `script`.

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

Si vous avez déjà rencontré des scripts externes sur le Web, vous aurez peut-être remarqué que les exemples de balises `script` plus anciens ont tendance à comporter des attributs spécifiques par défaut, comme `language` ou `type` :

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


`</script>`

Ces attributs sont obsolètes depuis HTML5. Il vaut mieux ne plus s'en préoccuper, et se contenter d'écrire `<script src="js/script.js"></script>`.

Naturellement, `script` accepte tous les attributs globaux d'HTML (`class`, `id`, `data-` et ainsi de suite), et HTML5 a ajouté quelques attributs utiles (et facultatifs) pour l'élément `script`, que nous aborderons dans un instant.

L'endroit où l'on choisit d'inclure les feuilles de styles externes dans un document – que l'on place `<link href="css/all.css" rel="stylesheet">` dans le `head` du document ou juste avant la balise `</body>` – ne fait pas grande différence : on les met donc par convention dans le `head`. Pour ce qui est de JavaScript, en revanche, il faut faire un peu plus attention au positionnement, que les scripts soient externes ou fassent partie de la page elle-même.

Placement de script

Au risque de trop simplifier, les navigateurs analysent le contenu d'un fichier de haut en bas. Lorsqu'on inclut un fichier de script en haut de la page, le navigateur analyse, interprète et exécute ce script avant de savoir quels éléments se trouvent sur la page. Ainsi, si l'on souhaite utiliser le DOM pour accéder à des éléments de la page, il faut donner au navigateur le temps d'assembler une carte de ces éléments en parcourant le reste de la page – autrement, lorsque notre script cherchera l'un de ces éléments, il produira une erreur indiquant que l'élément n'existe pas.

Plusieurs méthodes permettent de résoudre ce problème au sein du script lui-même – le navigateur dispose de méthodes pour informer JavaScript lorsque la page a été entièrement analysée, et nous pouvons demander à nos scripts d'attendre cet événement – mais il y a d'autres inconvénients à inclure des scripts au sommet d'une page.

À inclure trop de scripts dans le `head` de nos pages, celles-ci risquent en effet de s'en trouver ralenties. En effet, lorsqu'il rencontre un script externe dans le `head` du document, le navigateur arrête de charger la page le temps de télécharger et d'analyser le script, puis passe au script suivant (sauf intervention de notre part, les scripts s'exécutent toujours dans l'ordre d'apparition) ou charge la page elle-même. Si nous introduisons un grand nombre de scripts dans le `head` de notre document, nos utilisateurs risquent d'attendre un moment avant de voir la page apparaître.

Une alternative à ce retard de rendu et à ces erreurs potentielles consiste à

inclure les scripts en bas de la page, juste avant la balise `</body>`. Comme la page est analysée de haut en bas, on s'assure ainsi que notre balisage est prêt (et la page rendue) avant que nos scripts ne soient téléchargés.

Procéder ainsi évite de retarder le rendu de la page, mais repousse le téléchargement du script, ce qui n'est pas toujours idéal ; il faut parfois qu'un script soit chargé aussi vite que possible, avant même que le DOM ne soit disponible. Par exemple, Modernizr – une collection de scripts permettant de tester la prise en charge par le navigateur de fonctionnalités CSS et JavaScript – recommande que vous l'incluez dans le `head` du document afin que les résultats de ces tests soient immédiatement utilisables (<https://modernizr.com/>). Modernizr est suffisamment léger pour que le retard du rendu qu'il occasionne soit presque imperceptible, et les résultats de ses tests de fonctionnalités doivent être disponibles à l'intention de tous les autres scripts de la page, de sorte que la vitesse de chargement est essentielle. Il semble acceptable de bloquer le rendu de la page pendant une fraction de seconde pour s'assurer qu'elle fonctionne de manière fiable.

defer et async

Si HTML5 a supprimé beaucoup de vieux attributs obsolètes de l'élément `script`, il en a également ajouté quelques-uns pour gérer certains des problèmes évoqués ci-dessus : `<script async>` et `<script defer>`.

L'attribut `async` indique au navigateur qu'il doit exécuter le script de manière asynchrone (comme son nom l'indique). Lorsqu'il rencontre la balise `<script src="script.js" async>` au début du document, le navigateur initie une requête pour télécharger le script et le charge sitôt qu'il est disponible, mais poursuit le chargement de la page entre-temps. Cela résout le problème des scripts bloquant le rendu de la page dans le `head`, mais ne garantit toujours pas que la page sera chargée à temps pour pouvoir accéder au DOM ; par conséquent cet attribut n'est à utiliser que dans les situations où nous n'avons pas besoin d'accéder au DOM – ou lorsque nous précisons dans le code que nous souhaitons attendre que le document ait fini de se charger avant que notre script ne manipule le DOM. Ce qui nous amène un nouveau problème : si l'on charge plusieurs scripts en utilisant `async`, il devient impossible de savoir s'ils seront chargés dans leur ordre d'apparition. Il ne faut donc pas utiliser `async` pour des scripts qui dépendent les uns des autres.

Avec l'attribut `defer`, il n'est plus nécessaire d'attendre que le DOM soit disponible ; cet attribut indique au navigateur qu'il doit télécharger ces scripts, mais ne pas les exécuter avant d'avoir fini de charger le DOM. Avec `defer`, les scripts inclus au début du document sont téléchargés parallèlement

au chargement de la page elle-même, de façon à réduire le risque que l'utilisateur ne subisse un délai perceptible, et à empêcher l'exécution des scripts tant que la page n'est pas prête à être modifiée. Et contrairement à `async`, `defer` exécute nos scripts dans l'ordre dans lequel ils apparaissent.

Ces deux attributs règlent de façon pratique tous nos problèmes de requêtes bloquantes et de synchronisation, avec un petit bémol toutefois : si `defer` existe depuis longtemps, il n'a été que récemment standardisé, et `async` est tout nouveau, donc nous ne pouvons pas garantir sa compatibilité avec tous les navigateurs.

Dans le livre *Design web responsive et responsable* de la collection A Book Apart (paru en français aux éditions Eyrolles), Scott Jehl recommande de charger les scripts de façon asynchrone en utilisant JavaScript lui-même : un minuscule script de chargement placé dans le `head` du document télécharge les scripts supplémentaires au besoin (<http://bkaprt.com/jsfwd/01-01/>). Cela nous permet non seulement de charger des scripts de manière efficace et asynchrone, mais également de décider s'ils doivent être chargés ou non : si l'on détecte qu'un utilisateur est sur un appareil qui supporte les événements tactiles, par exemple, on peut charger un script qui définit des événements tactiles personnalisés pour notre interface. Si les événements tactiles ne sont pas pris en charge, inutile d'envoyer une requête pour ce script – et la requête la plus efficace est toujours celle que l'on n'envoie pas.

Cela fait déjà pas mal de choses à retenir pour ce qui est du chargement des scripts, et nous ne faisons qu'effleurer le sujet. Dans les exemples qui suivent, cependant, nos besoins seront simples. Nous voulons que la page soit complètement chargée avant que nos scripts ne soient exécutés, donc rien ne doit se trouver dans le `head` du document. Nous n'aurons donc pas besoin d'utiliser `defer`. Un script externe placé en dehors du `head` ne causera aucun blocage, aussi n'aurons-nous pas besoin d'`async` non plus. Comme il n'est pas vraiment nécessaire de bloquer le rendu de la page avec les scripts que nous allons écrire (et comme nous aurons besoin que le DOM soit disponible, par la suite), nous inclurons nos scripts externes juste avant la balise `</body>`.

Une toile vierge

Avant de pouvoir écrire du JavaScript sérieusement, nous avons besoin d'une toile vierge : un répertoire contenant un bon vieux document HTML.

```
<!doctype html>
<html lang="en">
<head>
```

```
<meta charset="utf-8">
</head>
<body>
</body>
</html>
```

À des fins de simplicité et de cohérence, nous allons charger notre script juste avant la balise `body` de fermeture. Comme il s'agit d'un script externe, notre élément `script` comportera un attribut `src` renvoyant vers notre fichier de script – qui, pour l'instant, n'est qu'un fichier vide intitulé `script.js`. J'enregistre généralement mes scripts dans un sous-répertoire intitulé `js/` ; ce n'est absolument pas obligatoire, mais cela peut aider à rester organisé.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
</head>
<body>
  <script src="js/script.js"></script>
</body>
</html>
```

Votre éditeur et vous

Comme avec HTML et CSS, débiter en JavaScript ne demande pas de grands moyens : n'importe quel éditeur de texte fera l'affaire. La syntaxe JavaScript peut être un peu plus dure à déchiffrer à l'œil nu, cependant, du moins jusqu'à ce que vous ayez baigné dedans pendant un certain temps. Pour mieux comprendre votre script au premier coup d'œil, utilisez un éditeur qui permet d'associer des couleurs aux mots-clés, fonctions, variables et ainsi de suite.

Heureusement, vous serez bien en peine de trouver un éditeur de code moderne qui ne gère pas la syntaxe JavaScript. Il y a fort à parier que votre éditeur préféré gère la coloration syntaxique de JavaScript tout aussi bien que l'HTML et CSS, simplement en reconnaissant l'extension `.js` – ce qui n'est pas évident pour nous à en voir notre fichier `script.js` vide.

Nous aurons absolument besoin de cette fonctionnalité lorsque nous commencerons à assembler des composants de JavaScript en scripts utiles et fonctionnels. Mais le temps d'apprendre les bases, contentons-nous d'ouvrir

notre navigateur préféré et de nous faire une première idée avec les outils de développement du navigateur.

NOS OUTILS DE DÉVELOPPEMENT

Il fut un temps – il n’y a pas si longtemps, à vrai dire – où les navigateurs ne nous étaient pas d’une grande aide avec JavaScript. Dans le meilleur des cas, le navigateur nous informait qu’il avait rencontré une erreur inconnue, incluant éventuellement le numéro approximatif de la ligne où se trouvait celle-ci. Pour déboguer un script, il fallait le modifier, recharger la page, espérer que rien n’explose et répéter le processus jusqu’à avoir débusqué l’erreur. Il n’y avait généralement aucun moyen d’obtenir plus de détails sur le problème, ou du moins de détails utiles.

Heureusement, nos outils de développement se sont améliorés au fil du temps, et les navigateurs de bureau modernes intègrent tous des outils de débogage JavaScript avancés. Nous pouvons toujours aborder le développement « à l’ancienne », bien sûr, mais c’est un peu comme utiliser la crosse d’un pistolet à clous pour clouer une planche, avec le risque de nous tirer dans le pied en prime.

Familiarisez-vous avec vos outils de développement et vous perdrez beaucoup moins de temps à traquer les bugs. Pour nos besoins, ceux-ci nous offrent un espace pour commencer à expérimenter avec la syntaxe de JavaScript.

Nous utiliserons ici les outils de développement de Chrome, mais les mêmes bases s’appliqueront à votre navigateur préféré. Les outils de développement des navigateurs sont généralement très similaires, jusqu’à la commande qui sert à les ouvrir : Commande+Option+I sur un Mac, Contrôle+Shift+I sur un PC. Certains détails vous paraîtront légèrement différents d’un navigateur à l’autre, mais vous constaterez qu’ils présentent tous un agencement similaire.

Si vous avez déjà utilisé ces outils pour inspecter des éléments et déboguer les problèmes de vos feuilles de styles CSS, vous connaissez l’onglet « elements », qui affiche tous les éléments de la page et les styles associés. Pour le reste, les autres onglets varieront un peu d’un navigateur à un autre.

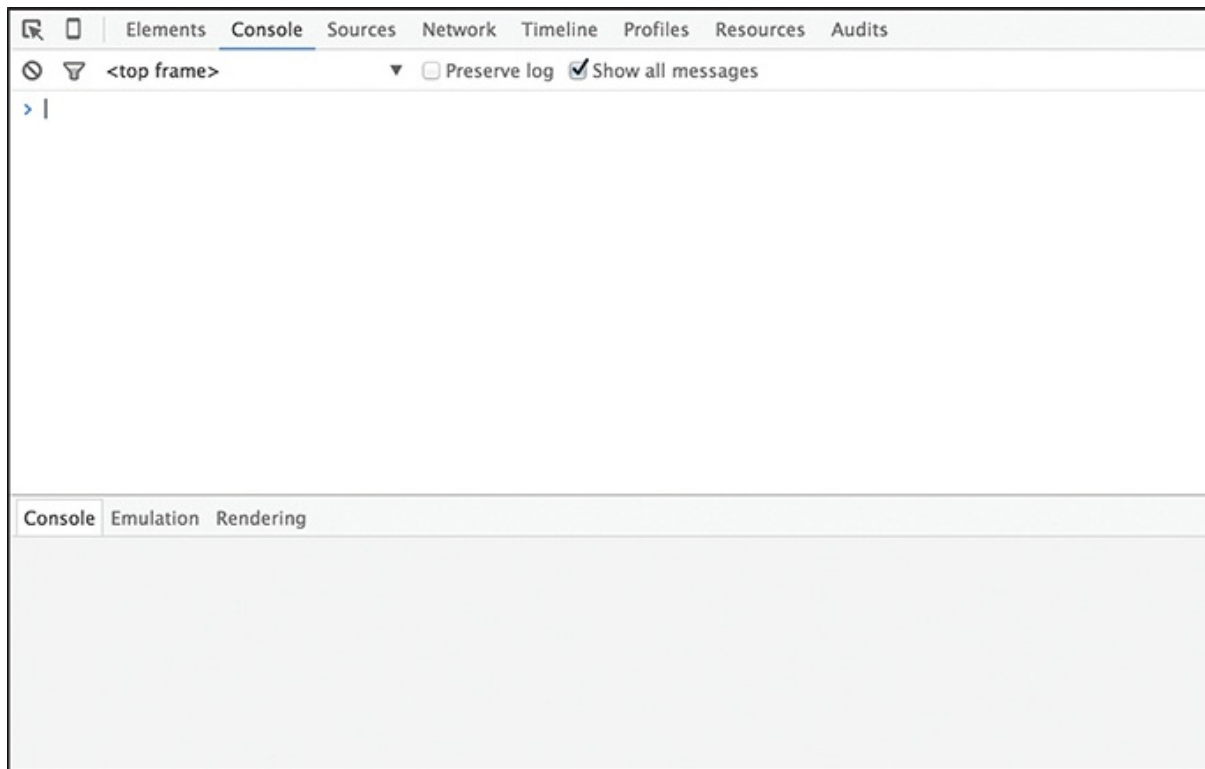


FIG 1.1 : Les outils de développement de Chrome s’ouvrent sur l’onglet « console ».

La plupart des outils de développement comportent également un onglet « network », qui permet de suivre le nombre et la taille des requêtes effectuées par une page et le temps passé à les charger, ainsi qu’un onglet « resources » permettant de parcourir toutes les ressources associées à une page, des feuilles de styles aux fichiers JavaScript en passant par les cookies. Certains navigateurs incluront également des informations sur le réseau sous l’onglet « ressources ». Il y aura généralement une sorte d’onglet « timeline » permettant de visualiser des informations sur le rendu de la page, comme le nombre de fois que le navigateur doit parcourir la page et « repeindre » tous les styles – il vous faudra peut-être recharger la page en gardant cet onglet ouvert pour voir quelque chose ([FIG 1.1](#)).

Nous travaillerons pour l’essentiel dans l’onglet « console », qui permet d’exécuter du code JavaScript dans ce que l’on appelle une REPL (*read-eval-print loop*) ou boucle d’évaluation (<http://bkaprt.com/jsfwd/01-02/>). Vous pouvez écrire du JavaScript dans la console et l’exécuter de la même façon que le ferait le moteur JavaScript du navigateur si le code se trouvait sur la page. Cela présente plusieurs avantages : tout d’abord, on peut commencer à jouer avec JavaScript sur la page de son choix, sans se soucier d’avoir un environnement de développement. Ensuite, tout ce qu’on écrit dans la console est conçu pour être éphémère : par défaut, les changements effectués avec JavaScript sont effacés quand vous rechargez la page.

La console JavaScript

La console JavaScript remplit deux fonctions essentielles pour nous aider à tester et à déboguer : elle consigne les erreurs et autres informations, et fournit une ligne de commande pour interagir directement avec la page et avec nos scripts.

Sous sa forme la plus simple, la console JavaScript sert à indiquer les erreurs de syntaxe de vos scripts ; si une coquille vient à se glisser dans votre script, ou si une partie du script référence une ressource inexistante, vous savez immédiatement ce qui empêche votre script de tourner.

La plupart des consoles de développement ne se contentent pas de signaler les erreurs les plus évidentes, mais donnent également des avertissements sur les fonctionnalités que les navigateurs sont susceptibles de supprimer prochainement, les requêtes rejetées, etc. Il est très rare que je développe sans avoir la console ouverte, par simple mesure de sécurité.

Cependant, il arrivera souvent que votre script ne comporte pas d'erreur manifeste, mais ne semble pas marcher totalement comme prévu. Vous aurez sans doute aussi besoin d'un moyen simple de vérifier que certaines parties d'un script sont exécutées, comme une grande partie de la logique est invisible. Dans ces cas, vous pourrez utiliser certaines méthodes intégrées du navigateur pour lancer un feu de détresse de temps en temps – pour vous envoyer des messages, inspecter le contenu de certaines variables, et semer des petits cailloux afin de suivre le cheminement du script.

Dans les temps anciens, nous employions de vieilles méthodes JavaScript intégrées pour le débogage de base – la trop dénigrée `alert()`, méthode qui fait apparaître une fenêtre modale native comportant le message de notre choix, saisi entre guillemets à l'intérieur des parenthèses, et un bouton OK pour la fermer (**FIG 1.2**).

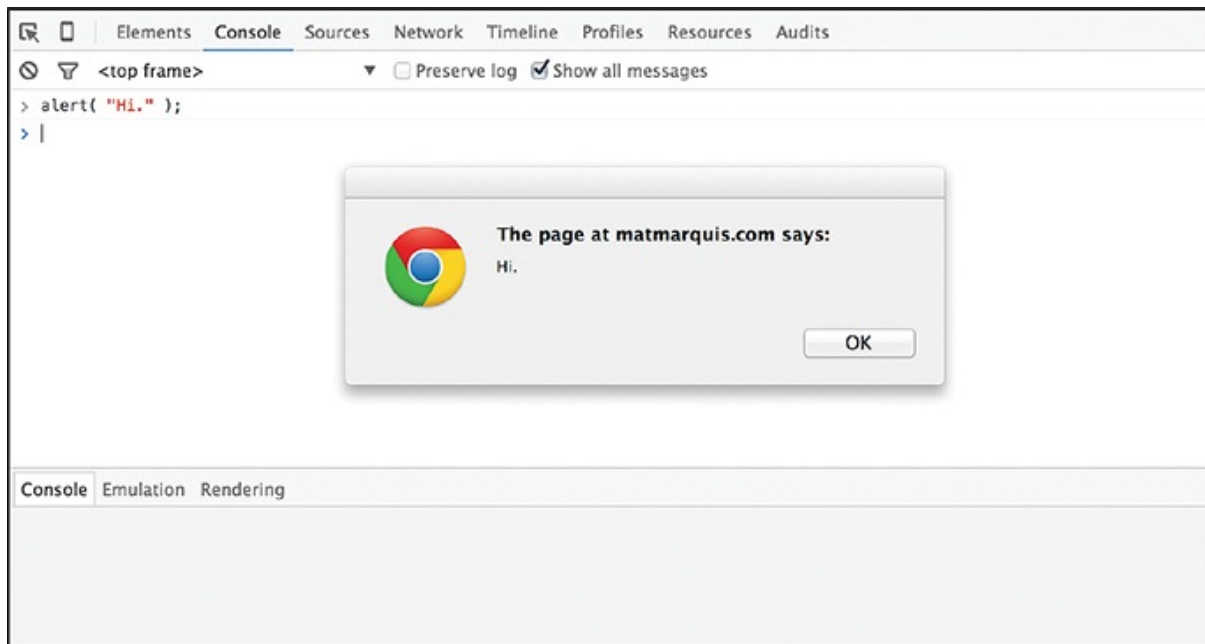


FIG 1.2 : Utiliser `alert()` sur un site web en ligne revient à crier « au feu » dans une salle de cinéma bondée : ce n'est pas forcément illégal, mais ça ne plaît pas à tout le monde.

Il existait plusieurs variations tout aussi palpitantes de cette méthode : `confirm()` offrait à l'utilisateur la possibilité de cliquer sur « OK » ou « annuler », et `prompt()` lui permettait de saisir du texte dans la fenêtre modale – ces deux méthodes rapportaient le choix ou le texte saisi par l'utilisateur pour que nous les utilisions dans la suite de notre script. Si vous avez déjà accepté un « contrat de licence utilisateur final » ou passé du temps sur Geocities à la grande époque, vous avez probablement déjà rencontré ces méthodes à un moment ou à un autre.

Les développeurs JavaScript ont rapidement compris qu'il s'agissait d'un moyen odieux d'interagir avec les utilisateurs, et aucune de ces méthodes n'est très utilisée ces jours-ci (ou alors, avec une certaine ironie).

Ce que nous avons appris, par ailleurs, c'est que ces méthodes constituaient un moyen simple de communiquer avec nous-mêmes à des fins de débogage, afin de mieux comprendre ce qui se passait dans nos scripts. Aussi inefficaces fussent-elles, nous pouvions programmer des `alert` pour déterminer jusqu'où notre script s'exécutait, ou pour vérifier que les différentes parties d'un script s'exécutaient dans le bon ordre, et (en voyant quelle était la dernière `alert` à s'afficher avant de rencontrer une erreur) pour retracer des erreurs manifestes ligne par ligne. Ce type de débogage ne nous en disait pas long, bien sûr – après tout, `alert` était seulement conçue pour transmettre une chaîne de texte, si bien que nous recevions souvent un feedback impénétrable comme `[object Object]` lorsque nous voulions nous intéresser de plus près à un morceau de

notre script.

Aujourd'hui, les outils de développement des navigateurs rivalisent de qualité et proposent des tonnes d'options pour explorer les rouages internes de nos scripts, la plus simple d'entre elles consistant toujours à nous envoyer un message depuis l'intérieur du script, mais contenant potentiellement beaucoup plus d'informations qu'une simple chaîne de texte.

Écrire dans la console

Le moyen le plus simple d'écrire quelque chose dans la console depuis notre script est une méthode appelée `console.log()`. Sous sa forme la plus simple, `console.log()` fonctionne comme `alert()` et permet de vous envoyer un message à travers votre script.

Une image vaut mieux qu'un long discours, alors ouvrons `script.js` dans notre éditeur et essayons la ligne suivante par nous-mêmes :

```
console.log("Hello, World.");
```

Sauvegardez le fichier, retournez dans votre navigateur, rechargez la page, et vous venez d'écrire votre toute première ligne de JavaScript (**FIG 1.3**).

Vous pensez peut-être qu'il n'y a pas de quoi s'exciter, mais `console.log` peut faire une grande partie du travail pour nous. Cette méthode présente elle aussi plusieurs variantes : nous pouvons utiliser `console.warn` et `console.error` de la même façon que `console.log`, afin de signaler des problèmes et des messages particuliers (**FIG 1.4**).

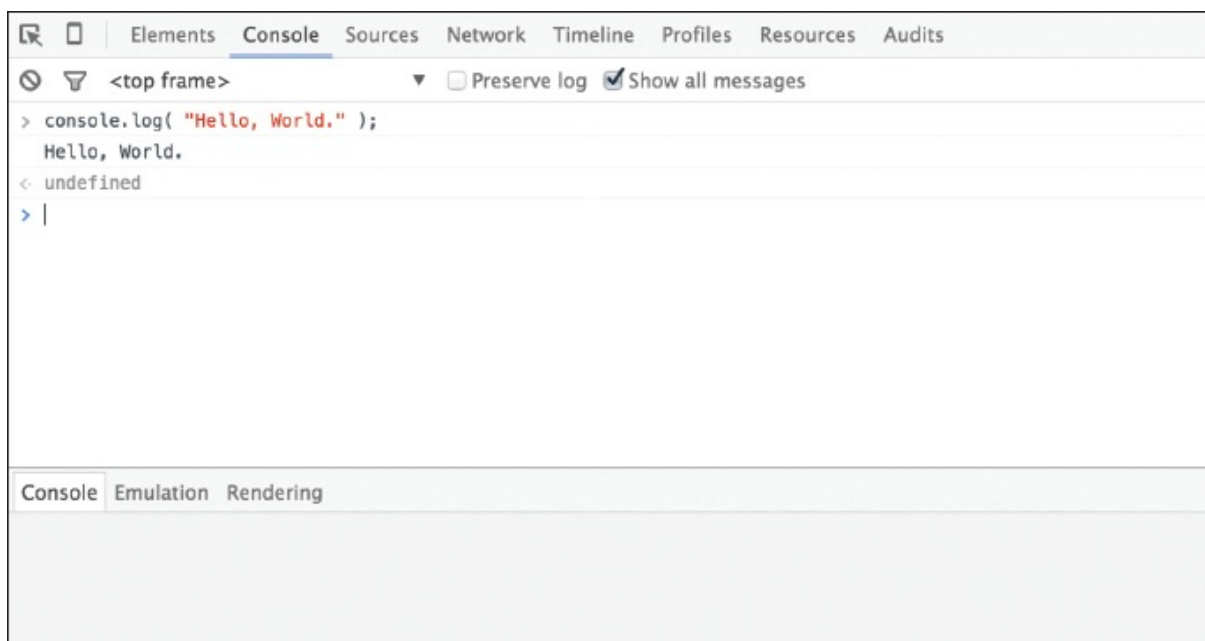


FIG 1.3 : Bien le bonjour à vous !

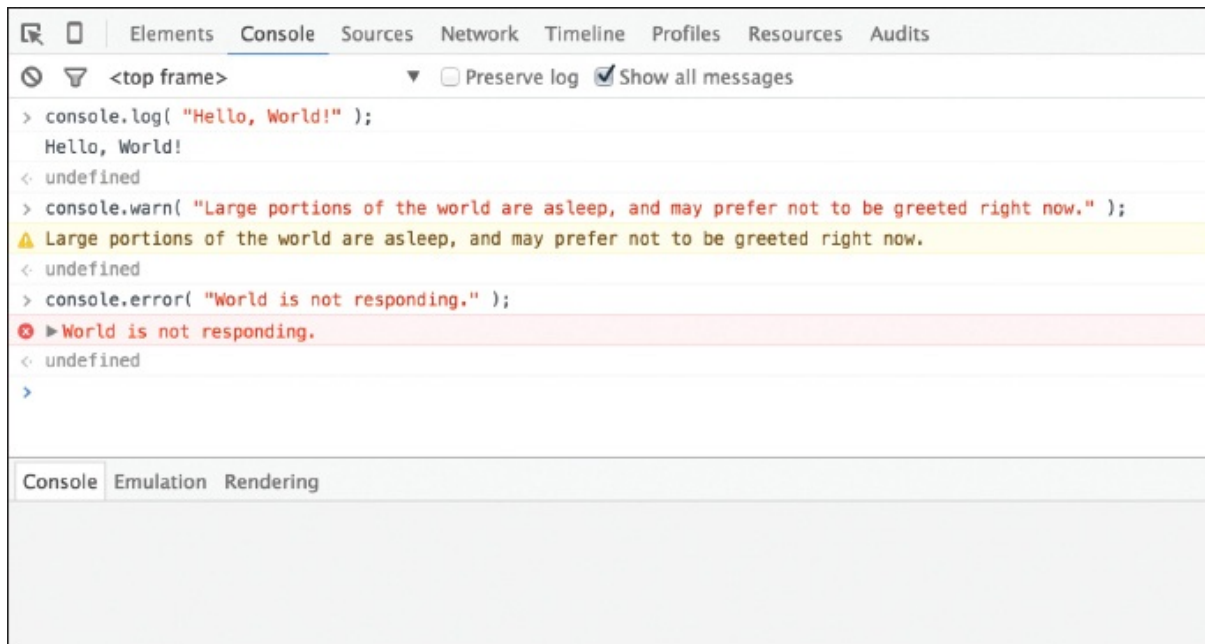


FIG 1.4 : Les méthodes `console.warn` et `console.error` sont toutes les deux utiles pour déboguer un script.

Une dernière remarque à propos de `console.log` et toute sa clique : si la plupart des navigateurs modernes permettent d'écrire dans une console, c'est loin d'être universel, et certains navigateurs plus anciens n'ont pas de console du tout. Internet Explorer 6 et 7, notamment, tombent en rade dès qu'ils rencontrent la méthode `console.log`.

Ces méthodes ne doivent pas apparaître dans le code définitif. Elles ne sont véritablement employées que pour écrire et déboguer nos scripts, donc elles ne risquent pas de causer des problèmes pour nos utilisateurs, sauf si nous en laissons une par accident. Veillez à vérifier qu'aucun code de débogage tel que `console.log` ne reste avant de mettre un script en ligne, par mesure de sécurité.

Travailler dans la console

La console JavaScript ne sert pas uniquement à tenir un registre des erreurs – vous aurez sans doute remarqué l'invite de commande clignotante. Cette invite est la boucle d'évaluation dont je parlais précédemment, la *read-evaluate-print-loop* (REPL).

Pour donner une brève définition de la REPL, celle-ci nous permet d'envoyer des commandes directement au parseur JavaScript du navigateur, sans avoir besoin de mettre à jour notre fichier de script et de recharger la page. Si vous saisissez la même commande `console.log("Hello, world.");` dans cet espace et que vous appuyez sur entrée, celle-ci s'affichera dans la console.

On peut utiliser cette invite pour obtenir des informations sur l'état actuel des éléments de la page, vérifier le produit d'un script, ou même ajouter une fonctionnalité sur la page pour la tester. Nous pouvons l'utiliser dès maintenant pour essayer de nouvelles méthodes et obtenir un feedback immédiat. Si vous tapez `alert("Test");` et que vous appuyez sur entrée, vous verrez ce que je veux dire : pas de fichier à changer, pas besoin de recharger la page, pas de tralala. Juste une bonne vieille fenêtre contextuelle bien moche.

Insistons sur le fait qu'on ne peut pas faire de vrais dégâts à travers la console : tous les changements que nous apporterons à la page ou à nos scripts par le biais de la console s'évaporeront sitôt la page rechargée, sans que nos fichiers n'aient été modifiés.

Nous avons désormais quelques options pour expérimenter avec JavaScript *via* la console, et un environnement de développement dans lequel nous pouvons commencer à bricoler nos premiers scripts ensemble. Nous sommes prêts à commencer à apprendre les règles de JavaScript.

LES RÈGLES FONDAMENTALES

JavaScript est complexe, cela va sans dire, mais les règles générales du langage sont étonnamment simples et généralement assez indulgentes, à quelques exceptions près. Il peut être bon d'aborder quelques-unes de ces règles au préalable ; ne vous inquiétez pas si elles ne vous paraissent pas tout à fait logiques tant que vous ne les aurez pas vues en action.

Sensibilité à la casse

Une règle stricte de JavaScript (qui me fait encore verser une larme de temps en temps), c'est qu'il fait la distinction entre majuscules et minuscules. Cela veut dire qu'`unevariable` et `uneVariable` sont traités comme deux éléments complètement différents. Cela devient problématique quand on s'aperçoit que les méthodes intégrées de JavaScript permettant d'accéder au DOM portent des noms comme `getElementsByTagName`, qui ne sont pas des plus simples à taper au clavier.

Pour l'essentiel, vous pouvez vous attendre à ce que les méthodes intégrées utilisent le CamelCase, capitalisant chaque mot sauf le premier, comme dans `querySelector` ou `getElementById`.

Vous pouvez observer cette règle en action dans la console : essayez de saisir `document.getElementById.toString();` et vous obtiendrez vraisemblablement une réponse décrivant l'usage de la méthode : le navigateur reconnaît cette phrase comme étant une méthode intégrée pour accéder à un élément dans le DOM. Mais si vous saisissez `document.getEleMentById.toString();`, avec le D de « Id » en majuscule, le navigateur renverra une erreur (**FIG 1.5**).

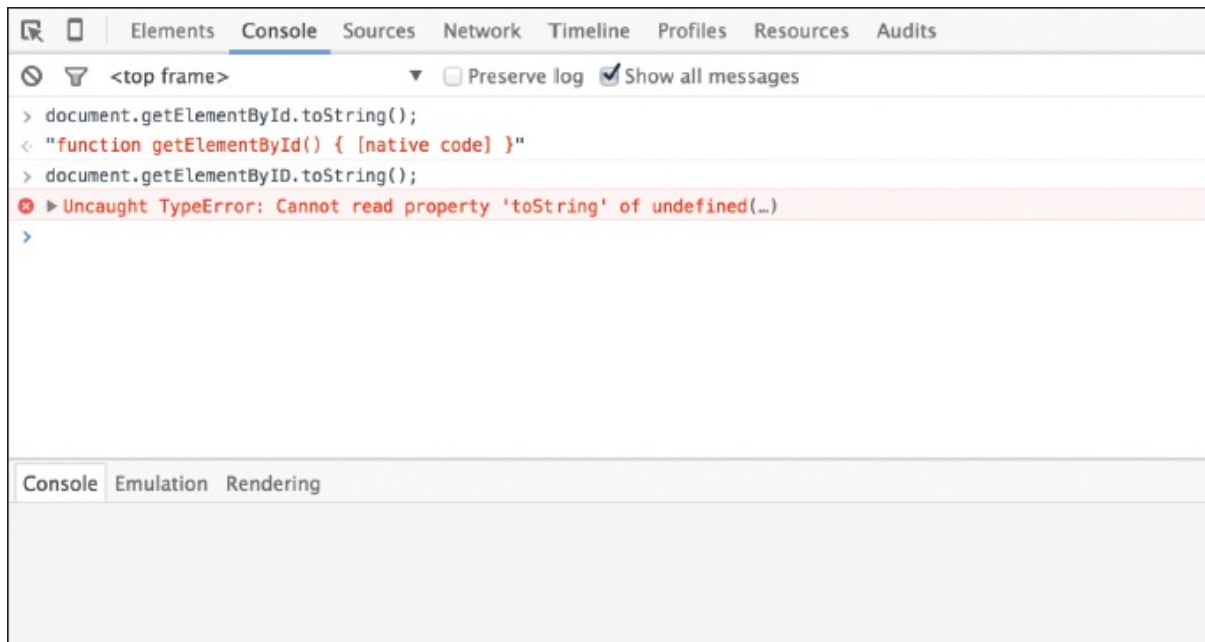


FIG 1.5 : Si proche, et pourtant...

Points-virgules

Une instruction en JavaScript doit pratiquement toujours se terminer par un point-virgule, qui indique au parseur JavaScript qu'il a atteint la fin d'une commande, de la même façon qu'un point conclut une phrase en français. Cette règle est quelque peu flexible : un saut de ligne signale également au parseur qu'il est arrivé à la fin de la déclaration, grâce à ce que l'on appelle l'insertion automatique de points-virgules (ou ASI, pour Automatic Semicolon Insertion).

Vous le savez peut-être, les programmeurs ont des opinions bien tranchées. Vous n'aurez pas à chercher bien longtemps pour trouver des débats interminables sur le thème « faut-il toujours utiliser un point-virgule à la fin d'une déclaration, ou plutôt gagner quelques octets et laisser ASI faire son travail ? ». Personnellement, je suis du premier camp ; je préfère être toujours explicite et mettre des points-virgules partout plutôt que de risquer d'en oublier un au mauvais endroit, sans compter que le code sera plus facile à lire pour la personne qui passera derrière moi pour l'entretenir. Pour commencer, je vous conseille de faire de même. Il vous faudra un certain temps avant de comprendre où les points-virgules sont strictement nécessaires et où ASI peut remplir les blancs, alors mieux vaut pécher par excès de prudence.

Espacement

Plus étrange encore, les sauts de ligne sont la seule forme d'espacement

(catégorie qui inclut aussi les tabulations et les espaces) qui ait une réelle signification pour JavaScript, et encore, seul le premier saut de ligne compte. Que vous utilisiez cinquante sauts de ligne entre chaque ligne de code, ou que vous commenciez chaque ligne par dix tabulations et une espace par superstition, JavaScript l'ignorera royalement. Seul le premier saut de ligne, qui indique à ASI que vous passez à la déclaration suivante, est véritablement important.

Commentaires

JavaScript permet d'écrire des commentaires qui seront ignorés lors de l'exécution du code, de sorte que vous pouvez laisser des notes et des explications au fil de votre code. Je trouve ça très utile au quotidien : je me laisse des indications et des pense-bêtes pour me rappeler pourquoi le code est organisé d'une certaine façon, ou bien des commentaires pour me rappeler qu'une partie du code doit être retravaillée, par exemple.

Mais surtout, vous devez savoir que vous ne serez pas toujours le seul propriétaire du code. Même si vous ne travaillez pas en équipe, il est fort probable qu'un jour, quelqu'un d'autre soit amené à modifier votre code. Un code bien commenté sert de feuille de route aux autres développeurs, et les aide à comprendre quelles décisions vous avez prises et pourquoi.

Il existe deux types de commentaires natifs en JavaScript, et vous reconnaîtrez l'un d'entre eux si vous touchez votre bille en CSS. Les commentaires multilignes sont gérés avec la même syntaxe qu'en CSS :

```
/* Ceci est un commentaire multiligne.  
Tout le texte qui se trouve entre ces balises sera ignoré  
lors de l'exécution du script. Ce type de commentaire doit  
être fermé. */
```

JavaScript permet également d'insérer des commentaires sur une seule ligne, qui n'ont pas besoin d'être explicitement fermés. Ils sont en effet fermés dès que vous démarrez une nouvelle ligne.

```
// Voici un commentaire sur une seule ligne.
```

Curieusement, les commentaires sur une seule ligne peuvent se poursuivre sur autant de lignes que nécessaire dans votre éditeur, du moment qu'ils ne contiennent pas de saut de ligne : dès que vous appuyez sur Entrée pour démarrer une nouvelle ligne, le commentaire est fermé et vous écrivez de nouveau du code exécutable. Cette particularité de votre éditeur de code (qui dépendra de l'éditeur lui-même et de vos réglages) s'appelle « soft wrapping ». Les commentaires sur une ligne ne seront pas affectés par le soft

wrapping, puisqu'il s'agit strictement d'une fonctionnalité de votre éditeur.

```
console.log("Hello, World."); // Note à moi-même :  
« World » prend-il un W majuscule ?
```


NOUS VOILÀ PRÊTS

Munis des règles du jeu et d'un espace pour expérimenter, nous sommes prêts à découvrir les composants de base de JavaScript. Ils vous paraîtront peut-être succincts au moment de s'asseoir devant l'ordinateur et d'écrire un script du début à la fin, mais le sujet que nous allons aborder dans le prochain chapitre est essentiel pour comprendre comment JavaScript traite les données.

2 COMPRENDRE LES TYPES DE DONNÉES

ATTENTION, ÇA VA SE CORSER.

D'ici la fin de ce chapitre, vous connaîtrez les types de données que vous rencontrerez au cours de votre épopée avec JavaScript. Certains types de données coulent de source, du moins à première vue : les nombres sont des nombres, les chaînes de texte sont des chaînes de texte. Certains types de données sont un peu plus ésotériques : `true`, un mot-clé en JavaScript, représente l'essence même de la Vérité.

Mais c'est un peu plus compliqué qu'il n'y paraît. Les nombres peuvent être *truthy* ou *falsy*, tandis que le texte sera toujours *truthy*. `NaN`, un mot-clé signifiant « pas un nombre » (*not a number*), est lui-même considéré par JavaScript comme un nombre. `({}+[])[!+[]+!+[]+!+[]]+(![]+[])[!+[]+!+[]+!+[]]` est une expression tout ce qu'il y a de plus valide en JavaScript. Vraiment.

On comprend facilement d'où JavaScript tient sa réputation de langage peu intuitif ; l'expression ci-dessus tient plus de la charade que du langage de programmation. Cependant, derrière cette folie se cache une méthode, et en apprenant à connaître les types de données de JavaScript, on peut commencer à penser comme JavaScript.

JavaScript est un langage « faiblement typé », ce qui signifie que l'on n'a pas besoin de déclarer explicitement qu'un élément doit être traité comme un nombre ou une chaîne de texte. Contrairement à un langage « fortement typé », qui exige de définir les données en précisant leur type, JavaScript est

capable de déduire le sens du contexte. Cela paraît logique, étant donné que la plupart du temps, il faut que 7 soit traité comme un nombre, et non comme une chaîne de texte.

Et si l'on tient à ce qu'un élément soit traité comme un type spécifique, JavaScript offre plusieurs moyens d'effectuer une conversion de type, afin de modifier l'interprétation des données. Heureusement, nous n'avons pas besoin de nous préoccuper de ça pour le moment – intéressons-nous plutôt aux types de données eux-mêmes.

TYPES PRIMITIFS

Certains types de données semblent évidents : on les appelle types primitifs. Les types primitifs sont réduits à leur expression la plus simple : un nombre est un nombre, `true` est vrai. Les primitifs sont les types de données les plus simples en JavaScript ; ils comptent `number`, `string`, `undefined`, `null`, ainsi que `true` et `false`.

Les nombres

Le type `number` représente l'ensemble de toutes les valeurs numériques possibles. JavaScript se débrouille plutôt bien avec les nombres, enfin, jusqu'à un certain stade. Si vous saisissez `7;` dans votre console et que vous appuyez sur entrée, le résultat ne devrait pas trop vous surprendre : ce sera 7. JavaScript reconnaît qu'il s'agit du chiffre sept. On fait des progrès de dingue aujourd'hui, vous ne trouvez pas ?

OPÉRATEUR	DESCRIPTION	USAGE	RÉSULTAT
<code>+</code>	Addition	<code>2+2</code>	<code>4</code>
<code>-</code>	Soustraction	<code>4-2</code>	<code>2</code>
<code>*</code>	Multiplication	<code>2*5</code>	<code>10</code>
<code>/</code>	Division	<code>10/5</code>	<code>2</code>
<code>++</code>	Ajouter 1 à un nombre	<code>2++</code>	<code>3</code>
<code>--</code>	Soustraire 1 à un nombre	<code>3--</code>	<code>2</code>
<code>%</code>	Calculer le reste d'une division	<code>12 % 5</code>	<code>2</code>

FIG 2.1 : Il n'y aura pas de quizz, c'est promis.

Le type générique `number` recouvre quelques cas particuliers : la valeur « pas un nombre » (`NaN`), ainsi qu'une valeur représentant l'infini, qui peut être positive (`Infinity`) ou négative (`-Infinity`). Si vous tapez l'une de ces valeurs dans votre console et que vous appuyez sur entrée, elle sera renvoyée par JavaScript tout comme le chiffre 7 – c'est la façon qu'a JavaScript de dire « je reconnais ce concept ». En revanche, si vous essayez de saisir `infinity;` ou `Nan;`, la console renverra `undefined` ; souvenez-vous, JavaScript est sensible aux majuscules.

Dans le même esprit, vous ne devriez pas être surpris de pouvoir utiliser des opérateurs mathématiques en JavaScript, qui font exactement ce qu'on

pourrait attendre d'eux. Si vous tapez `2+2;` dans votre console, JavaScript renverra 4.

Bon nombre de ces opérateurs vous seront familiers, même si vous n'avez que de maigres souvenirs de vos cours de maths au lycée ; certains d'entre eux sont plus spécifiques à la programmation (**FIG 2.1**).

L'ordre des opérations mathématiques s'applique ici ; toutes les expressions entre parenthèses sont évaluées en premier, suivies des exposants, des multiplications, des divisions, des additions et des soustractions. Je parie que vous ne vous attendiez pas à réentendre l'acronyme PEMDAS un jour. Ça ne nous rajeunit pas...

```
2*2-2;
```

```
2
```

```
2*(2-2);
```

```
0
```

Vous avez peu de chances de rencontrer *Infinity* au cours de votre carrière avec JavaScript, ou alors accidentellement. Si vous essayez de taper `2/0;` dans la console, en espérant que votre ordinateur ne soit pas happé par un trou noir, JavaScript renverra la valeur *Infinity*.

NaN est un cas spécial que l'on voit un peu plus fréquemment. À chaque fois qu'on tente de traiter des données non numériques comme des nombres, JavaScript renvoie *NaN*. Par exemple, si l'on prend la phrase `"Hello, world."` avec laquelle nous avons expérimenté la méthode `console.log`, et qu'on la multiplie par deux (`"Hello, world." * 2`), on obtient *NaN* comme résultat. JavaScript ne sait pas ce que vous êtes censé obtenir en multipliant un mot par un nombre, mais il sait avec certitude qu'il ne s'agit pas d'un nombre.

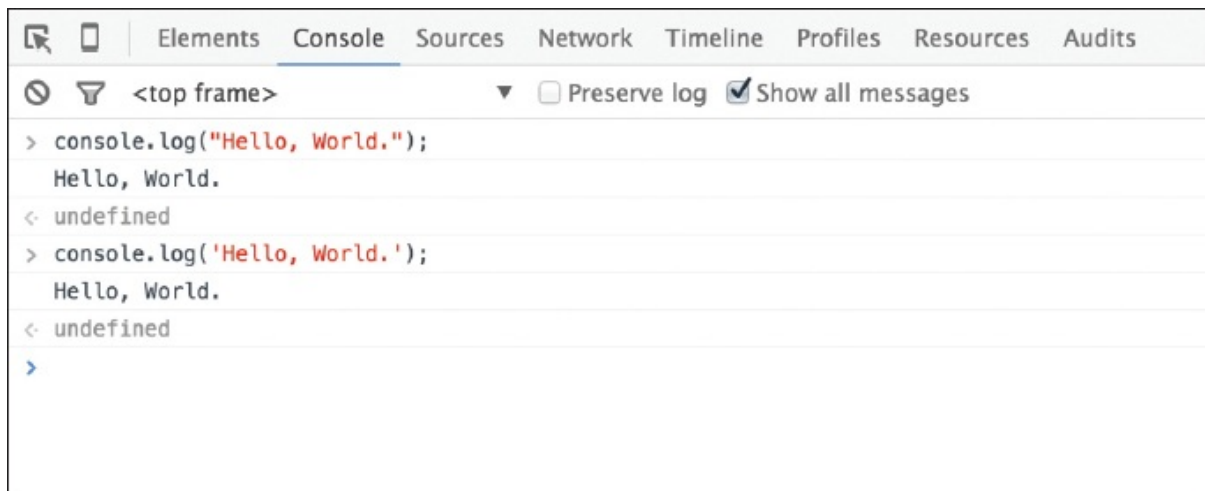
Les chaînes

Les chaînes de texte sont probablement le type de données le plus simple à comprendre. Tout ensemble de caractères – lettres, chiffres, symboles, etc. – placé entre des guillemets simples ou doubles est considéré comme une « chaîne ».

D'ailleurs, nous avons déjà vu une chaîne de caractères : lorsque nous avons écrit `console.log("Hello, world.");` dans la console au chapitre précédent, `"Hello, world."` était la chaîne. Nous obtiendrions le même résultat avec des guillemets simples, à savoir `console.log('Hello, world.');` (**FIG 2.2**).

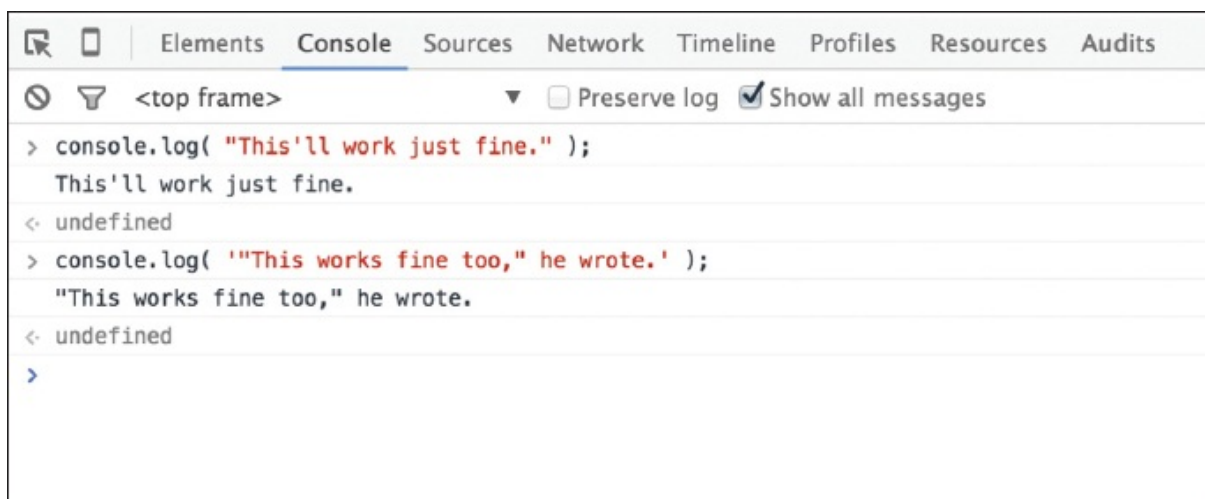
Les guillemets simples et doubles sont fonctionnellement identiques, du

moment que vous les appariez correctement, et une chaîne utilisant des guillemets doubles peut également contenir des guillemets simples, ou vice-versa ([FIG 2.3](#)).



```
> console.log("Hello, World.");  
Hello, World.  
< undefined  
> console.log('Hello, World.');
```

FIG 2.2 : Les guillemets simples et doubles produisent exactement le même résultat.



```
> console.log( "This'll work just fine." );  
This'll work just fine.  
< undefined  
> console.log( '"This works fine too," he wrote.' );  
"This works fine too," he wrote.  
< undefined  
>
```

FIG 2.3 : Tout va bien, du moment que nos guillemets sont correctement associés.

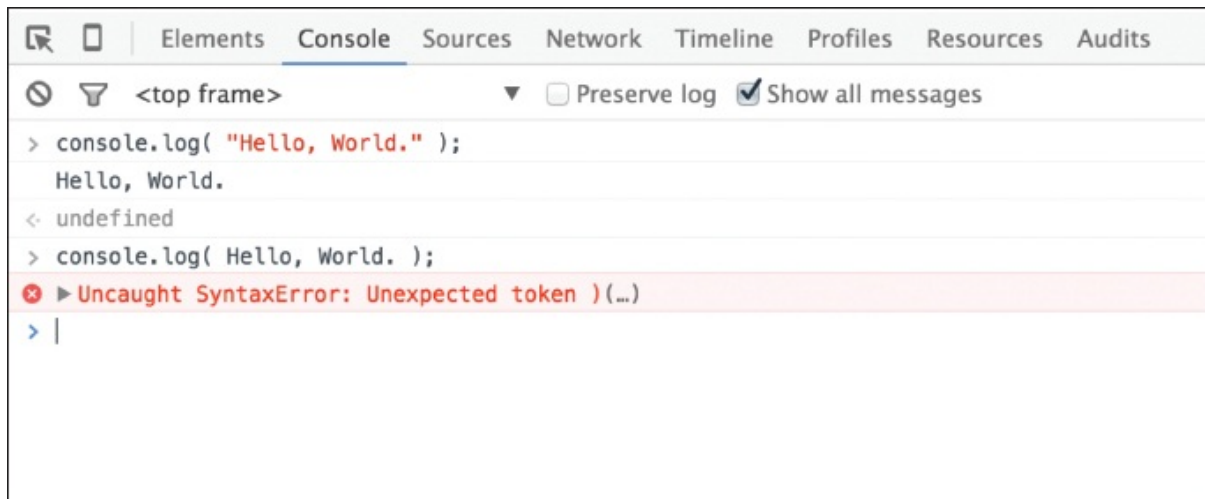


FIG 2.4 : Oups...

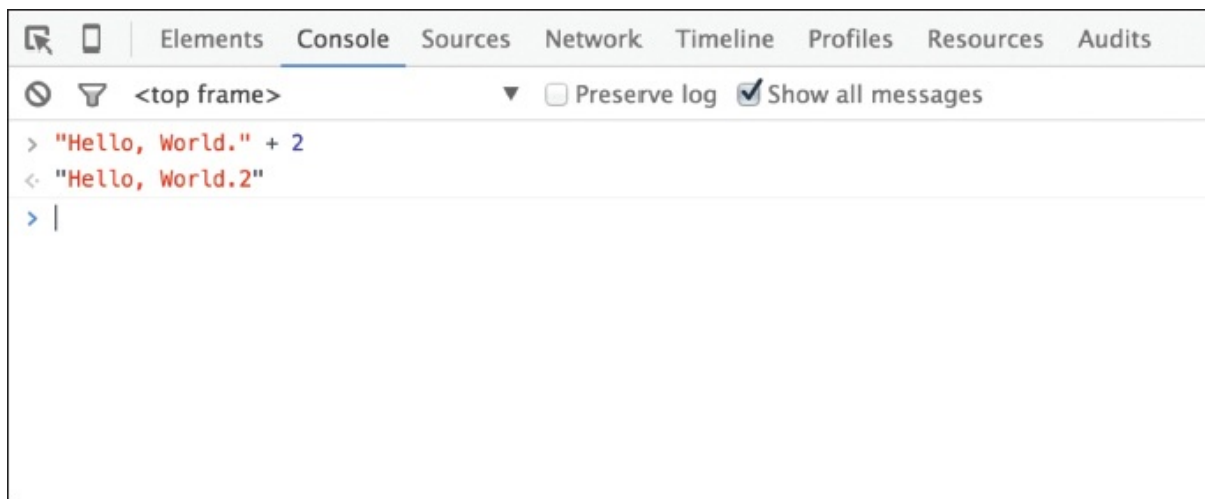


FIG 2.5 : La suite est toujours moins bonne que l'original.

En revanche, si l'on omet les guillemets les résultats sont très différents. Sans les guillemets, JavaScript essaie de lire `Hello, world.` comme un élément du script plutôt que comme une chaîne de texte, ce qui cause une erreur de syntaxe (FIG 2.4).

Les chaînes sont remarquablement simples : quelques lettres et/ou chiffres entre guillemets. Mais elles possèdent une propriété importante : elles peuvent être assemblées entre elles.

L'assemblage de plusieurs chaînes s'appelle concaténation. Vous pouvez réunir deux chaînes ou plus à l'aide du signe plus, qui fait double office, à la fois pour les additions mathématiques et la concaténation de chaînes, selon le contexte dans lequel il est utilisé (FIG 2.5).

Lorsqu'on travaille avec des chaînes plutôt que des nombres, le signe `+` n'essaie pas d'exécuter une addition mathématique. Au lieu de cela, il concatène deux types de données en une seule chaîne. Bien que l'exemple ci-

dessus comporte un chiffre, la présence d'une chaîne de caractères conduit JavaScript à traiter également `2` comme une chaîne.

undefined

Comme on pourrait s'y attendre, `undefined` est le type de données utilisé pour tout ce qui n'est pas prédéfini par JavaScript, ou par nous dans le cadre de notre script. Vous avez déjà vu des exemples de ce type de données : en jouant avec la casse dans nos consoles de développement, nous avons saisi plusieurs commandes non reconnues par JavaScript et obtenu une erreur en retour (**FIG 2.6**).

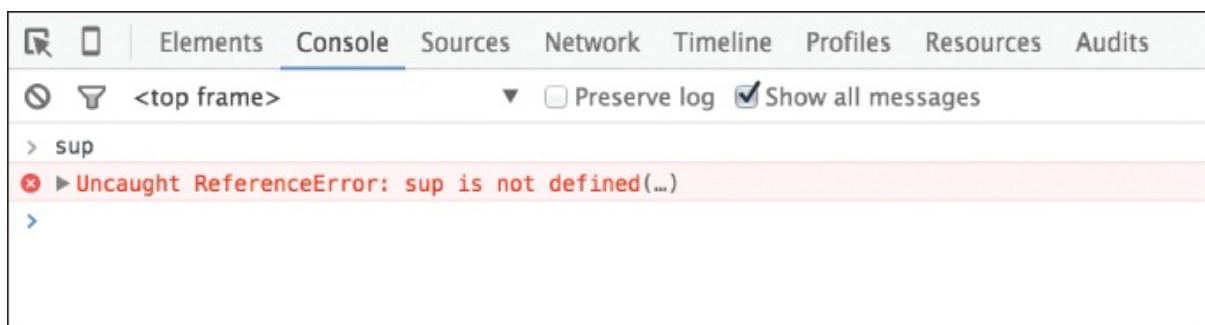


FIG 2.6 : Nous n'avons pas dit à JavaScript que `sup` voulait dire quelque chose, et il connaît mal l'argot.

Si l'on utilise `typeof` – un opérateur qui renvoie le type d'un opérande non évalué sous la forme d'une chaîne – pour déterminer le type de `sup`, nous verrons que son type est `undefined`. Pour JavaScript, `sup` n'a aucun sens ni aucune valeur ; nous ne lui en avons jamais donné.

null

`null` représente une non-valeur : un élément a été défini, mais il n'a aucune valeur intrinsèque. Par exemple, on peut définir une variable comme étant `null` dans l'attente qu'elle se voie attribuer une valeur à un certain point du script, ou affecter la valeur `null` à une référence existante pour effacer la valeur précédente.

Booléens

Les valeurs booléennes – les mots-clés `true` et `false` – représentent le vrai et le faux. Vous retrouverez ces concepts dans tous les langages de programmation. Si l'on demande à JavaScript de comparer deux valeurs et qu'elle s'avèrent égales, l'expression renverra la valeur `true` – et dans le cas

contraire, *false*.

Revenons à notre console, et ce faisant, regardons la vérité absolue dans le fond des yeux :

```
2 + 2 == 4;
```

```
true
```

Et tant que nous y sommes, essayons un peu de novlangue orwellienne :

```
2 + 2 == 5;
```

```
false
```

Bon, d'accord, ce n'est pas aussi dramatique que je l'aurais voulu, mais ces types de comparaisons sont fondamentaux pour la logique de vos scripts.

Notez que nous utilisons `==` pour effectuer une comparaison, plutôt que le `=` auquel vous auriez pu vous attendre : pour JavaScript, le signe égal sert à attribuer une valeur à quelque chose, et non à effectuer une comparaison entre deux valeurs. Nous y reviendrons dans un instant – et nous aborderons les opérateurs de comparaison dans la suite de ce chapitre.

TYPES D'OBJETS

Le concept d'« objet » en JavaScript est à rapprocher du concept d'objet dans le monde réel. Dans les deux cas, un objet est une collection de propriétés et de méthodes – c'est-à-dire de traits appartenant à un objet et de fonctions que l'objet peut remplir. Dans le monde réel, par exemple, l'objet « marteau » est l'abstraction de propriétés (« manche », « surface de frappe lourde ») et d'une finalité (« frapper des choses »). Mais le concept de marteau est modifiable : si l'on modifie ses propriétés (« osselet de l'oreille ») et sa finalité (« transmettre le son »), « marteau » peut signifier quelque chose de complètement différent.

Un objet JavaScript est similaire : c'est une collection de propriétés et de méthodes modifiables qui porte un nom. En dehors des types primitifs listés ci-dessus, chaque morceau de JavaScript que nous écrivons est un « objet », des chaînes de texte et des nombres que nous définissons jusqu'au document tout entier.

Cela peut sembler incommensurable, mais les types d'objets spécifiques que nous sommes susceptibles de rencontrer au quotidien sont clairement différenciés.

Variables

Une variable est un nom symbolique représentant une valeur. Comme autant de x dans un cours d'algèbre de collège, les variables peuvent contenir des données de tous types : chaînes de texte, nombres, éléments rapatriés *via* le DOM, voire des fonctions entières. Une variable constitue un point de référence unique pour cette valeur, à utiliser dans tout ou partie du script. Nous pouvons modifier la valeur de cette variable à l'envi.

Il existe deux manières de déclarer une variable, employant toutes deux le signe égal simple, qui effectue une affectation au lieu d'une comparaison. La manière la plus simple de déclarer une variable n'utilise pas grand-chose d'autre : on spécifie l'identifiant, puis on utilise le signe `=` pour lui attribuer une valeur.

```
foo = 5;
```

5

Lorsque l'on crée une variable pour la première fois, la console répond en renvoyant la nouvelle valeur de la variable.

Désormais, si l'on saisit `foo`; et qu'on appuie sur entrée, on obtient le même résultat : une variable appelée `foo` a été déclarée et on lui a donné la valeur 5. Une fois la variable définie, son comportement est identique aux données qu'elle contient. Voyons ce qui se passe si l'on vérifie le type de la variable `foo` à l'aide de l'instruction `typeof` :

```
foo = 5;
5
foo;
5
typeof foo;
"number"
```

Le type de `foo` est maintenant « number », et non « variable ». Pour JavaScript, la variable `foo` est fonctionnellement identique au chiffre 5. Cependant, cet état n'est pas définitif : on peut réutiliser une variable en lui attribuant une nouvelle valeur.

```
foo = 5;
5
foo = 10;
10
```

Nous pouvons même réattribuer une valeur à une variable en utilisant la variable elle-même :

```
foo = 100;
100
foo = foo * foo;
10000
```

Bien sûr, nous ne saurons pas toujours à l'avance quelle valeur la variable devra contenir. Après tout, le principe de base, c'est que les variables peuvent représenter n'importe quelle valeur dans un emballage prévisible et facile à référencer. Même si l'on ne veut pas donner de valeur initiale à la variable, on peut la déclarer en JavaScript. L'instruction `var foo`; permet de déclarer une nouvelle variable (`foo`) comme étant `undefined`, aussi bizarre que cela puisse paraître. JavaScript identifie désormais le mot « foo » comme une variable, mais sans qu'aucune valeur ne lui soit attribuée. Testez dans votre console JavaScript et vous verrez ce que je veux dire.

```
var bar;
undefined
bar;
```

```
undefined
```

```
blabla;
```

```
Uncaught ReferenceError: blabla is not defined
```

Nous avons défini `bar` comme une variable ; ainsi, lorsque nous saisissons son nom dans notre console, la boucle d'évaluation renvoie sa valeur comme un bon perroquet docile. Cette valeur, comme nous ne lui en avons pas donné une, est *undefined*. Si nous essayons la même chose avec une variable que nous n'avons pas définie – `blabla`, en l'occurrence – JavaScript nous pond une erreur.

Vous avez remarqué le mot-clé `var` ? Il n'est pas strictement nécessaire d'utiliser `var` pour déclarer une variable si vous lui attribuez immédiatement une valeur, mais pour des raisons que j'aborderai prochainement, il est toujours judicieux de déclarer vos variables avec l'instruction `var`. De même qu'il est préférable de toujours clore l'attribution d'une variable par un point-virgule, même si JavaScript ne l'exige pas.

```
var foo = 5;
```

```
undefined
```

Ne vous inquiétez pas si la console renvoie *undefined* après l'attribution d'une valeur à une variable – le moteur JavaScript n'a rien à répondre à des tâches telles que la déclaration de variables, et renvoie donc la valeur *undefined* en retour.

Il est également possible de déclarer plusieurs variables à la fois. Comme souvent en JavaScript, il existe deux options pour définir plusieurs variables, employant deux syntaxes différentes, mais équivalentes. La première utilise une seule instruction `var` et scinde la liste des variables et des données associées par des virgules (en finissant par un point-virgule, bien sûr) :

```
var foo = "hello",
```

```
bar = "world",
```

```
baz = 3;
```

La seconde méthode consiste à utiliser plusieurs instructions `var` séparées :

```
var foo = "hello";
```

```
var bar = "world";
```

```
var baz = 3;
```

Aucun piège dans ce cas. Ces deux syntaxes fonctionnent exactement de la même façon, et le choix de l'une ou de l'autre est entièrement une question de préférence personnelle. Et naturellement c'est un sujet très controversé dans les cercles de développeurs JavaScript.

Bien sûr, il serait irresponsable pour moi d'imposer mon opinion personnelle à vous, lecteur, alors je me contenterai de dire ceci : respectez toujours les conventions de programmation existantes d'un projet, plutôt que de mélanger plusieurs méthodes. Sur un projet tout nouveau, utilisez la syntaxe que vous trouvez la plus confortable, mais gardez l'esprit ouvert – nous avons trop de problèmes complexes à résoudre pour se chamailler sur des histoires de préférence personnelle. Et en cas de doute, faites comme moi, parce que j'ai toujours raison.

Non, vraiment, faites ce qui vous semble le plus pratique.

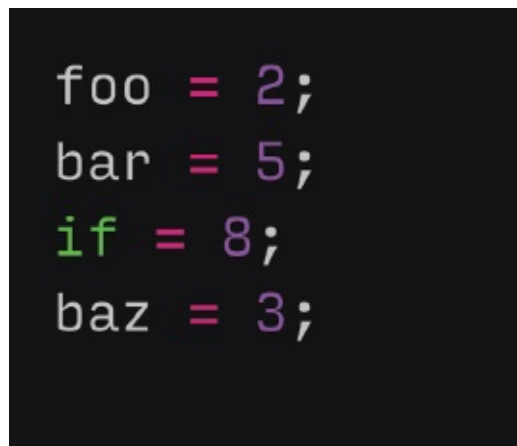
Mais dans tous les cas, j'ai raison.

Identifiants

Le nom que l'on donne à une variable est un identifiant.

Comme tout le reste en JavaScript, les identifiants sont sensibles à la casse et s'accompagnent de quelques règles spéciales :

- Ils doivent commencer par une lettre, un tiret bas ou le signe dollar – pas par un chiffre.
- Ils ne peuvent pas contenir d'espaces.



```
foo = 2;
bar = 5;
if = 8;
baz = 3;
```

FIG 2.7 : La coloration syntaxique permet de repérer plus facilement les erreurs à la volée.

- Ils ne peuvent pas contenir de caractères spéciaux (! . , / \ + - * =).

Il y a toute une liste de mots qui ne peuvent pas être utilisés comme identifiants en JavaScript, comme `null`, par exemple. C'est ce que l'on appelle des mots-clés – des mots déjà utilisés en JavaScript, ou réservés au cas où ils y seraient ajoutés un jour :

```
abstract boolean break byte case catch char class const
continue debugger default delete do double else enum export
```

```
extends false final finally float for function goto if
implements import in instanceof int interface long native new
null package private protected public return short static super
switch synchronized this throw throws transient true try typeof
var void volatile while with
```

Cela fait un sacré paquet de mots, mais rien ne vous oblige à les retenir par cœur ; pour ma part, je m'en suis abstenu. En revanche, c'est là qu'un éditeur avec coloration syntaxique prend toute son importance, en vous évitant de rencontrer de mystérieuses erreurs lorsque vous attribuez un identifiant à une variable (**FIG 2.7**).

En dehors de ces règles, un identifiant peut contenir n'importe quelle combinaison de lettres, de chiffres et de tirets. Il est préférable d'utiliser des identifiants courts (`coutTotal` plutôt que `valeurDeTousLesArticlesTaxesEtLivraisonIncluses`) et faciles à comprendre du premier coup d'œil (`valeurSelectionnee` vs `v1`). Les `foo`, `bar` et autres `baz` que j'ai utilisés dans mes exemples sont de piètres identifiants – des mots qui n'ont aucun sens et ne fournissent aucun indice sur la nature des données qu'ils contiennent. À l'inverse, mieux vaut éviter les identifiants qui décrivent leurs valeurs potentielles de façon trop détaillée, car il n'est pas toujours possible de prédire les valeurs qu'une variable contiendra. Une variable appelée à l'origine `miles` pourra un jour être appelée à contenir une valeur en kilomètres – déroutant pour les développeurs qui devront gérer ce code, comme pour nous. L'identifiant `distance` serait beaucoup plus approprié.

Portée des variables

Nous verrons ce point plus en détail en abordant les fonctions, mais nous ne pouvons pas parler des variables sans évoquer ce que l'on appelle la portée des variables.

Voyez la portée des variables comme la zone de votre code source au sein de laquelle vous avez attribué un identifiant à un élément. En dehors de cette zone, cette variable n'est pas définie, et l'identifiant peut être réutilisé pour autre chose. Les applications JavaScript peuvent être énormes, avec des dizaines de milliers de lignes de code à analyser et à exécuter. Les variables ayant chacune leur propre portée, on peut choisir de les mettre à la disposition de l'application tout entière ou de les restreindre à certaines sections du code, pour éviter que des centaines de variables n'encombrent notre application. S'il fallait tenir un inventaire mental des identifiants qui sont déjà utilisés pour ne pas risquer de réutiliser ou redéfinir accidentellement une variable, nous aurions vraiment besoin de l'un de ces cerveaux de programmeur-cyborg dont nous parlions au début de ce livre.

Une variable peut avoir deux portées : locale et globale. Une variable définie en dehors d'une fonction est globale. Et comme leur nom l'indique, les variables globales sont accessibles dans la totalité de l'application.

Une variable définie à l'intérieur d'une fonction peut être locale ou globale, selon la façon dont nous la définissons – en l'occurrence, selon que nous la déclarons avec l'instruction `var` ou non. À l'intérieur d'une fonction, `var` déclare une variable dans le champ d'application local de cette fonction, mais si l'on omet `var`, cela signifie que cette variable doit être globale, ou en d'autres termes, exposée à l'application tout entière.

```
(function() {  
  var foo = 5;  
})();  
undefined  
console.log( foo );  
Uncaught ReferenceError: foo is not defined
```

La portée des variables est un sujet compliqué, et nous en aborderons les détails plus délicats lorsque nous étudierons les fonctions. Pour l'instant, retenez simplement qu'il est toujours préférable de définir vos variables à l'aide de `var`. En utilisant systématiquement `var`, les variables locales seront locales et les variables globales resteront globales – ce qui nous évitera de passer des heures à chercher la fonction qui a modifié la valeur d'une variable globale de manière imprévue. Et lorsque le temps sera venu d'exposer une variable locale au champ d'application global, nous verrons de meilleures façons d'y parvenir que d'omettre `var`.

Tableaux

Les tableaux (*arrays*) sont très semblables aux variables, à une exception près : si une variable ne peut contenir qu'une seule valeur, un tableau peut en contenir plusieurs, comme une liste. D'autre part, la syntaxe est similaire à celle des variables :

```
var monPremierTableau = [ "item1", "item2" ];  
undefined
```

Cela doit vous rappeler des souvenirs : une instruction `var`, suivie d'un identifiant de votre choix, puis d'un signe égal pour attribuer une valeur. Les mêmes règles s'appliquent pour l'identifiant là aussi – d'ailleurs, toutes les règles des variables s'appliquent aux tableaux, y compris la portée.

Pour le reste, cependant, quelques différences sont à noter : au lieu que

l'identifiant désigne un seul type de données, on crée une liste – dans cet exemple, une paire de chaînes – insérée entre des crochets et dont les composants sont séparés par des virgules. Sachez simplement que les espaces à l'intérieur du tableau n'ont aucune importance, mais sont simplement une question de préférence personnelle. `var monPremierTableau = ["item1","item2"];` est 100 % identique, pour JavaScript, à `var monPremierTableau=["item1","item2"];`. Je trouve juste la première option un peu plus facile à lire.

De même que les variables, les tableaux peuvent se composer de tous types de données :

```
var monPremierTableau = [ "item1", 2, 3, true, "dernier  
    item" ];  
undefined
```

Quand on saisit cet identifiant dans la console de développement, celle-ci renvoie la valeur, tout comme avec une variable :

```
var monPremierTableau = [ "item1", 2, 3, true, "dernier  
    item" ];  
undefined  
monPremierTableau;  
["item1", 2, 3, true, "dernier item"]
```

Nous n'aurons sans doute pas besoin d'accéder à tout le tableau à la fois. Il est plus probable que ce tableau nous serve à emballer plusieurs données apparentées, dans le but d'y accéder individuellement, ce que nous ferons à l'aide d'indices : des numéros correspondant à leur position dans le tableau.

```
var monDeuxiemeTableau = [ "premier", "deuxième",  
    "troisième" ];  
undefined  
monDeuxiemeTableau;  
["premier", "deuxième", "troisième"]  
monDeuxiemeTableau[ 0 ];  
"premier"  
monDeuxiemeTableau[ 1 ];  
"deuxième"  
monDeuxiemeTableau[ 2 ];  
"troisième"
```

Vous remarquerez que JavaScript balaie une hypothèse facile d'entrée de jeu :

on pourrait s'attendre à ce que le premier élément du tableau corresponde à l'indice **1**, mais JavaScript démarre l'indexation à partir de **0**.

Quand on référence une position dans un tableau à l'aide d'un indice, on en fait un usage similaire à celui d'une variable : toute référence à une position dans le tableau adopte le type de données de la donnée qu'elle contient – et comme avec une variable, on peut réaffecter une donnée à une position donnée dans le tableau à l'aide du signe égal.

```
var monDeuxiemeTableau = [ "premier", "deuxième",  
    "troisième" ];  
monDeuxiemeTableau[ 2 ];  
"troisième"  
typeof monDeuxiemeTableau[ 2 ];  
"string"  
monDeuxiemeTableau[ 2 ] = 3;  
3  
monDeuxiemeTableau;  
["premier", "deuxième", 3]  
typeof monDeuxiemeTableau[ 2 ];  
"number"  
monDeuxiemeTableau[ 3 ] = "numero cuatro";  
"numero cuatro"  
monDeuxiemeTableau;  
["premier", "deuxième", 3, "numero cuatro"]
```

Jusqu'ici, nous avons uniquement utilisé des crochets pour initialiser nos tableaux, et il faudra toujours utiliser des crochets pour accéder aux informations d'un tableau, mais il existe une méthode alternative pour initialiser un tableau :

```
var monPremierTableau = new Array( "item1", "item2" );  
undefined  
monPremierTableau;  
["item1", "item2"]
```

Comme vous le voyez, il revient au même d'utiliser des crochets ou la syntaxe `new Array()`.

De même, on peut utiliser des crochets ou la syntaxe `new Array()` pour initialiser un tableau sans éléments définis, tout comme on peut initialiser une variable et la laisser vide. Pour cela, nous pouvons utiliser des crochets vides,

ou la syntaxe `new Array()` sans rien écrire entre les parenthèses :

```
var tableauTrois = [];  
undefined  
var tableauQuatre = new Array();  
undefined
```

Là encore, ces deux syntaxes sont fonctionnellement identiques : elles permettent toutes deux d'initialiser un tableau vide.

Il y a une chose que la syntaxe `new Array()` est seule à pouvoir faire : initialiser un tableau avec un nombre d'éléments fixe, même quand les éléments eux-mêmes ne sont pas définis :

```
var tableauTroisElements = new Array( 3 );  
undefined  
tableauTroisElements;  
[undefined × 3]
```

Tout cela signifie qu'un nouveau tableau a été créé et qu'il contient trois éléments non définis pour le moment. Par ailleurs, le comportement de ce tableau est identique à ceux que nous avons vus jusqu'à présent : vous n'êtes pas limité à ces trois éléments, et vous pouvez définir les informations et y accéder exactement de la même manière.

Cette syntaxe peut s'avérer un peu délicate, cependant : avec `new Array()`, vous communiquez le nombre d'éléments que contient votre tableau de la même façon que vous spécifiez les données que vous souhaitez qu'il contienne. Vous pouvez donc obtenir des résultats très différents de la syntaxe entre crochets lorsque vous stockez des données numériques. JavaScript est suffisamment malin pour savoir que plusieurs nombres saisis entre les parenthèses de `new Array()` signifient que vous créez un ensemble de nombres :

```
var tableauNombres = [ 777, 42, 13, 289 ];  
undefined  
tableauNombres;  
[777, 42, 13, 289]  
var autreTableauNombres = new Array( 777, 42, 13, 289 );  
undefined  
autreTableauNombres;  
[777, 42, 13, 289]
```

Mais si vous cherchez à initialiser un tableau contenant un seul élément et

qu'il s'agit d'une donnée numérique, vous obtiendrez des résultats très différents avec les deux syntaxes. La notation entre crochets fonctionne normalement, et produit un tableau contenant un seul élément qui porte la valeur que nous lui avons attribuée :

```
var tableauNombres = [ 777 ];  
undefined  
tableauNombres;  
[777]  
tableauNombres[ 0 ];  
777
```

Avec la syntaxe `new Array()`, les choses deviennent un peu bizarres. Nous nous retrouvons avec un tableau contenant sept cent soixante-dix-sept éléments non définis.

```
var autreTableauNombres = new Array( 777 );  
undefined  
autreTableauNombres;  
Array[777]  
autreTableauNombres[ 0 ];  
undefined
```

Pour être franc, je n'ai jamais eu besoin d'initialiser un tableau avec un nombre donné d'éléments `undefined` d'entrée de jeu. Votre expérience sera peut-être différente, bien sûr, mais la notation entre crochets me convient parfaitement.

Une fois définis, les tableaux s'accompagnent d'un certain nombre de méthodes pour parcourir et modifier leurs données. Par exemple, la propriété `.length` sur un tableau décrit le nombre d'éléments qu'il contient :

```
var leTableauFinal = [ "premier élément", "deuxième  
    élément", "troisième élément" ];  
undefined  
leTableauFinal.length;  
3
```

Et comme l'indice lui-même est une bonne vieille donnée numérique, nous pouvons accéder aux informations d'un tableau de manière un peu plus créative :

```
var leTableauFinal = [ "premier élément", "deuxième  
    élément", "troisième élément" ];
```

undefined

```
// Rapporter le dernier élément du tableau :  
leTableauFinal[ leTableauFinal.length - 1 ];  
"troisième élément"
```

Nous utilisons ici la propriété `.length` du tableau pour trouver l'indice du dernier élément. Comme un tableau peut faire n'importe quelle longueur, nous ne pouvons pas utiliser un numéro pour accéder au dernier élément. Nous pouvons en revanche utiliser la propriété `.length` pour obtenir le décompte de tous les éléments du tableau ; cependant, comme l'indexation en JavaScript commence à zéro, nous ne pouvons pas juste utiliser la longueur du tableau. Rien de plus simple : il nous suffit de soustraire un de la longueur du tableau (un type de données numérique) pour obtenir l'indice du dernier élément.

Objets et propriétés

Un objet peut contenir plusieurs valeurs sous la forme de propriétés. Contrairement à un tableau, qui contient un ensemble de données d'un certain type et affecte un indice numéroté à chaque élément, les propriétés d'un objet sont nommées à l'aide de chaînes de caractères.

```
var monChien = {  
  "nom" : "Zéro",  
  "couleur" : "orange",  
  "ageEnAnnees" : 3.5,  
  "bienEleve" : false  
};
```

undefined

Chaque propriété se compose d'un couple clé/valeur. La « clé » est une chaîne que l'on définit et qui renvoie à une valeur – de la même façon qu'on nomme les variables, il faut que les clés portent un nom prévisible, flexible et facile à comprendre. Dans l'exemple ci-dessus, les clés de chaque propriété de l'objet `monChien` s'appellent `nom`, `couleur`, `ageEnAnnees` et `bienEleve`, et leurs valeurs respectives sont les chaînes `Zéro` et `orange`, le nombre `3.5` et la valeur booléenne `false`.

Les propriétés d'un objet elles-mêmes peuvent être traitées comme des objets ayant leurs propres propriétés, ce qui nous permet de regrouper une énorme quantité d'informations sous un format très compact.

```
var monChien = {
```

```
"nom" : {  
  "prenom" : "Zéro",  
  "secondPrenom" : "Baskerville",  
  "nomDeFamille" : "Marquis"  
},  
"couleur" : "orange",  
"ageEnAnnees" : 3,  
"bienEleve": false  
};  
undefined
```

Souvenez-vous que l'espacement utilisé dans ces exemples (l'indentation, les sauts de ligne et les espaces autour des deux-points) n'a aucune importance pour JavaScript. Il sert uniquement à rendre le code plus lisible.

Définir un objet

Il existe deux façons de définir un nouvel objet. L'une consiste à utiliser l'instruction `new`, dont la syntaxe ne devrait pas vous être complètement étrangère à ce stade :

```
var monChien = new Object();  
undefined
```

La seconde consiste à employer une notation utilisant un littéral objet :

```
var monChien = {};  
undefined
```

Ces deux syntaxes fonctionnent essentiellement comme la déclaration d'une variable : l'instruction `var` est suivie d'un identifiant et du signal égal.

Ces deux méthodes permettant de définir un objet fonctionnent de la même façon, à une grosse différence près : l'instruction `new` nous oblige à définir d'abord un objet, puis à le remplir de données.

```
var monChien = new Object();  
undefined  
monChien.surnom = "Falsy";  
"Falsy"
```

La notation utilisant un littéral objet nous permet de définir et d'affecter des données à un objet simultanément.

```
var monChien = {
```

```
"surnom": "Falsy"
};
undefined
```

Vous verrez que beaucoup de développeurs favorisent la notation à littéral objet pour des raisons de simplicité, et nous ferons de même dans ce chapitre et les suivants.

Manipulation des propriétés

Une fois que nous avons défini un objet à l'aide de l'une des deux méthodes décrites ci-dessus, nous avons deux moyens d'accéder aux informations contenues dans un objet et de les modifier : la notation à point et la notation à crochets.

Pour accéder aux informations contenues dans la propriété d'un objet à l'aide de la notation à point, vous devez placer un point entre l'identifiant de l'objet et la clé de la propriété.

```
var monChien = {
  "nom": "Zéro"
};
undefined
monChien.nom;
Zéro
```

La notation à crochets utilise une paire de crochets et une chaîne désignant la clé à laquelle on veut accéder, tout comme on utiliserait un indice dans un tableau. Contrairement à la notation à point, on utilise une chaîne de caractères pour référencer les clés – et il faut donc placer la propriété `nom` entre guillemets.

```
var monChien = {
  "nom": "Zéro"
};
undefined
monChien[ "nom" ];
Zéro
```

Si la notation à crochets requiert une chaîne de caractères, c'est précisément la raison pour laquelle elle existe à la base : dans les scripts complexes, il faut parfois accéder à certaines clés en se basant sur la logique personnalisée que nous avons codée. Pour cela, nous aurons parfois besoin de créer une chaîne

personnalisée à partir de chaînes de caractères, de nombres, de variables, etc.

Admettons que notre script tire aléatoirement l'une des clés dans l'objet suivant :

```
var voitures = {  
  "voiture1" : "rouge",  
  "voiture2" : "bleue",  
  "voiture3" : "verte"  
};  
undefined
```

Nous pouvons créer une variable contenant un chiffre compris entre un et trois, et l'utiliser pour créer une chaîne qui renvoie à l'une de ces trois clés. Il existe de nombreuses manières de générer un nombre aléatoire avec JavaScript, mais pour simplifier l'exemple, nous utiliserons simplement le chiffre 2, en créant une chaîne concaténée appelée `voiture2`.

```
var voitures = {  
  "voiture1" : "rouge",  
  "voiture2" : "bleue",  
  "voiture3" : "verte"  
};  
undefined  
var cleVoiture = "voiture" + 2;  
undefined  
cleVoiture;  
"voiture2"  
voitures.cleVoiture;  
undefined
```

Nous ne pourrions pas utiliser la notation à point dans une situation comme celle-ci, car JavaScript ne traiterai pas `cleVoiture` comme une variable. Étant donné le fonctionnement de la notation à point, JavaScript pense que `cleVoiture` est l'identifiant de la clé que nous cherchons, et non pas la chaîne qu'elle contient.

La notation à crochets, à l'inverse, s'attend à une chaîne de caractères – et comme c'est justement ce que `cleVoiture` contient, le code suivant fonctionne parfaitement :

```
var voitures = {  
  "voiture1" : "rouge",
```

```
    "voiture2" : "bleue",
    "voiture3" : "verte"
};
undefined
var cleVoiture = "voiture" + 2;
undefined
cleVoiture;
"voiture2"
voitures[ cleVoiture ];
"bleue"
```

Vous découvrirez de nombreux moyens astucieux d'employer la notation à crochets au fil de votre carrière de « JavaScripteur ». Mais si vous n'avez pas besoin d'être astucieux, je trouve la notation à point bien plus simple à lire et à utiliser.

Fonctions

Une fonction est un bloc de code réutilisable qui nous permet de réaliser des tâches répétitives sans reproduire le même code à travers le script. Au lieu de cela, on utilise un identifiant pour référencer une fonction qui contient ce code, et on communique à la fonction toute information dont elle a besoin pour exécuter une tâche à notre place.

En bref, une fonction est un objet qui fait quelque chose, plutôt que de simplement contenir une valeur.

Définir une fonction implique un peu plus de code que ce que nous avons vu jusqu'à présent, mais la première partie ne devrait pas trop vous surprendre. Comme d'habitude, `var` définit la portée, puis nous spécifions un identifiant de notre choix, et nous utilisons le signe égal pour affecter une valeur à cet identifiant. Au lieu d'une simple chaîne de caractères, d'un nombre ou d'une valeur booléenne, nous faisons suivre le signe égal de l'instruction `function` et d'une paire de parenthèses. Ensuite, entre deux accolades, nous plaçons tout le code que nous voulons que cette fonction exécute quand nous y faisons appel. Comme d'habitude, nous clôturons la déclaration par un point-virgule.

```
var quoideneuf = function() {
    console.log( "Hello again, world." );
};
undefined
```


Si l'on colle ce code dans la console, rien ne se produit. Pas de *Hello again, world.*, ou du moins, pas encore. Pour l'instant, nous avons simplement défini une fonction avec l'identifiant `quoideneuf`, qui, lorsqu'elle sera appelée, produira la phrase "Hello again, world."

Si vous tapez `quoideneuf`; dans votre console, celle-ci renverra soit *function quoideneuf()*, soit l'intégralité du code de la fonction, selon le navigateur ; dans les deux cas, le navigateur déclare connaître une fonction qui porte cet identifiant. Pour véritablement exécuter cette fonction, nous devons l'appeler en utilisant l'identifiant et une paire de parenthèses :

```
var quoideneuf = function() {  
    console.log( "Hello again, world." );  
};  
undefined  
quoideneuf;  
function quoideneuf()  
quoideneuf();  
Hello again, world.
```

Dans leur forme la plus simple, les fonctions ne semblent peut-être pas terriblement utiles, car on a rarement besoin d'exécuter exactement les mêmes lignes de code (qui produiraient le même résultat). Le vrai pouvoir des fonctions réside dans le fait que vous pouvez leur faire passer des informations qui seront interprétées par le code, produisant ainsi des résultats différents. Les parenthèses qui suivent l'identifiant de la fonction ne demandent pas seulement au navigateur d'exécuter la fonction associée : elles peuvent servir à faire passer des informations au code situé à l'intérieur de la fonction, sous la forme d'arguments.

```
var bienvenue = function( username ) {  
    console.log( "Rebonjour, " + username + "." );  
};  
undefined
```

En ajoutant `username` entre parenthèses lors de la définition de la fonction, on indique que la fonction doit créer une variable nommée `username`, et que cette variable doit contenir la valeur spécifiée entre parenthèses lors de l'exécution de la fonction. Dans ce cas, la fonction s'attend à une chaîne de caractères qui sera concaténée avec notre message de bienvenue :

```
var bienvenue = function( username ) {  
    console.log( "Rebonjour, " + username + "." );
```

```
};  
undefined  
bienvenue( "Wilto" );  
Rebonjour, Wilto.
```

Nous ne faisons rien pour valider les données transmises à la fonction (nous y viendrons par la suite), mais pour l'instant, la concaténation de chaîne est suffisamment robuste, grâce à la conversion de type de JavaScript. Même si nous spécifions un type de données différent, les choses fonctionnent généralement comme prévu.

```
var bienvenue = function( username ) {  
    console.log( "Rebonjour, " + username + "." );  
};  
undefined  
bienvenue( 8 );  
Rebonjour, 8.  
bienvenue( true );  
Rebonjour, true.
```

Forcément, des problèmes font surface si nous omettons tout bonnement l'argument :

```
var bienvenue = function( username ) {  
    console.log( "Rebonjour, " + username + "." );  
};  
undefined  
bienvenue();  
Rebonjour, undefined.
```

Comme nous n'avons affecté aucune valeur à `username`, mais que JavaScript reconnaît l'identifiant, `username` est un type de données non défini. Grâce à la conversion de type, le type de données `undefined` est devenu la chaîne « undefined ». Ce n'est pas la façon la plus élégante de formuler les choses, mais ce n'est pas la plus inexacte non plus : après tout, la fonction salue quelqu'un dont nous n'avons jamais défini le nom.

L'un des usages les plus courants (et les plus puissants) des fonctions consiste à les utiliser comme une méthode réutilisable et bien conditionnée permettant de calculer quelque chose. Je n'entends pas par là un calcul strictement mathématique, même si c'est tout à fait possible. En programmant une fonction pour qu'elle « renvoie » une valeur, nous lui permettons d'être traitée de la même façon que nous traiterions une variable : comme un contenant de

données qui se comporte tout comme les données qu'il contient.

Les fonctions peuvent potentiellement renvoyer, et se comporter comme, le résultat final d'une logique infiniment complexe, plutôt que comme des données définies à la main. Nous n'aborderons les logiques infiniment complexes qu'au chapitre suivant, alors pour l'instant, nous demanderons à notre fonction de renvoyer quelque chose de relativement simple : la somme de deux valeurs.

```
function addition( num1, num2 ) {  
    return num1 + num2;  
}  
undefined  
addition( 4, 9 );  
13  
typeof addition( 2, 2 );  
"number"
```

Pour aller un peu plus loin, nous pouvons même attribuer la valeur renvoyée par une fonction à une variable :

```
function addition( num1, num2 ) {  
    return num1 + num2;  
}  
undefined  
var somme = addition( 2, 3 );  
undefined  
somme;  
5  
typeof somme;  
"number"
```

Il est important de se souvenir que si l'on utilise l'instruction `return`, le but final de la fonction consiste à renvoyer une valeur. Si nous incluons du code dans une fonction après une instruction `return`, ce code ne sera jamais exécuté.

```
function concatChaines( premiereChaine, deuxiemeChaine ) {  
    return premiereChaine + deuxiemeChaine;  
    console.log( "Est-ce que quelqu'un m'entend ?" );  
}
```

```
undefined  
concatChaines( "Chaînes", " test" );  
"Chaînes test"
```

Comme l’instruction `return` est placée avant `console.log`, l’instruction `console.log` ne sera jamais exécutée. En fait, il se peut même que votre éditeur la surligne comme étant une erreur.

Tout ou presque est un objet

Voilà pour les types d’objets courants que vous rencontrerez au fil de vos aventures avec JavaScript. Bien que nous ne les ayons pas encore utilisés de manière particulièrement complexe, une fois combinés, ceux-ci constituent l’intégralité de JavaScript. Au bout du compte, JavaScript se compose d’outils prédéfinis qui se comportent exactement comme ceux avec lesquels nous avons joué jusqu’à présent.

Sous certaines conditions, tout, sauf `null` et `undefined`, peut être considéré comme un objet – même les chaînes de caractères, qui sont probablement le type de données le plus simple de tous. Une nouvelle chaîne comprend des méthodes et des propriétés intégrées, comme les tableaux, même si nous nous contentons de définir un petit bout de texte :

```
"test".length;
```

4

Techniquement, cette chaîne en elle-même n’est pas un objet ; elle ne comporte aucune méthode ni propriété propre. Mais lorsque l’on demande la valeur de la propriété `length`, JavaScript sait de quoi on parle, car il a une liste de méthodes et de propriétés prédéfinies qu’il applique à toutes les chaînes.

Est-ce une distinction nécessaire ? Non, pas à ce stade : d’ailleurs, ça risquerait même de vous induire en erreur. Mais à mesure que vous découvrirez les caractéristiques de JavaScript, cette distinction commencera à prendre tout son sens. En attendant, vous verrez sans doute ce comportement décrit beaucoup plus succinctement lorsque le sujet de ce qui est (ou n’est pas) un objet JavaScript sera abordé : « tout est un objet... en quelque sorte ».

Maintenant que nous avons découvert certains des composants de base qui composeront nos scripts, nous pouvons commencer à les utiliser pour dérouler une logique. En d’autres termes : maintenant que nous connaissons les bases, nous pouvons commencer à écrire des scripts qui font plus qu’afficher aveuglément du texte dans notre console.

Par défaut, un navigateur « lit » un script de la même façon que vous liriez une page en français : de gauche à droite et de haut en bas. Des instructions de contrôle du flux permettent de contrôler quelles portions de notre code sont exécutées à tel ou tel moment, voire si elles sont exécutées tout court.

Cela semble compliqué au début, mais tout peut être découpé en une poignée de déclarations très simples qui permettent, une fois combinées, de faire des choses incroyablement complexes. Pour nos besoins, les instructions de contrôle du flux peuvent être divisées en deux catégories : les expressions conditionnelles et les boucles. C'est ce que nous allons aborder dans les deux prochains chapitres.

3

LES EXPRESSIONS CONDITIONNELLES

LES EXPRESSIONS CONDITIONNELLES sont des instructions de contrôle du flux qui gèrent la logique : elles déterminent quand et comment exécuter du code, selon les conditions que vous avez définies.

INSTRUCTIONS **if/else**

Les expressions conditionnelles sont pratiquement toutes construites sur une variation du type « si X, faire Y ». L'exemple le plus courant de cette logique – qui est présent dans tous les langages de programmation – est la structure **if/else**. « Si ci, fais ça » est une expression logique des plus simples, mais d'ici la fin de chapitre, vous verrez comment quelque chose d'aussi simple en surface peut constituer la majeure partie de la logique de nos scripts.

if

Sous sa forme la plus simple, une instruction **if** exécute le code défini entre des accolades, mais seulement si le contenu des parenthèses qui suivent le mot **if** s'avère être vrai (**true**).

Dans le précédent chapitre, nous avons vu que JavaScript renvoyait *true* pour l'expression `2 + 2 == 4`, lorsque nous la saisissions dans notre console de développement. Au lieu de nous contenter d'une expression aussi simple, essayons notre première déclaration **if**. Souvenez-vous qu'un signe égal simple (=) sert à affecter des valeurs, tandis qu'un signe égal double (==) sert à effectuer une comparaison.

```
if( 2 + 2 == 4 ) {  
    console.log( "Salut !" );  
}  
  
Salut !
```

Rien de très surprenant ici : le texte « Salut ! » s'affiche dans notre console. Si l'on écrit une expression fausse entre parenthèses, la ligne contenant l'instruction `console.log` – ainsi que tout le code placé entre ces accolades – ne sera pas exécutée.

```
if( 2 + 2 == 5 ) {  
    console.log( "Salut !" );  
}  
  
undefined
```

Évidemment, ça ne paraît pas très utile si l'on utilise des expressions dont on sait déjà si elles sont vraies ou fausses, mais l'instruction **if** ne sert pas seulement à vérifier de temps à autre que les règles des mathématiques s'appliquent toujours. Si l'on considère que les objets JavaScript peuvent contenir toutes sortes de données complexes à manipuler de différentes façons

à travers un script, et que les objets sont traités comme les données qu'ils contiennent, on peut prendre des décisions particulièrement complexes quant au flux du script en utilisant de simples instructions `if`. Pour l'instant, nous allons simplement initialiser une variable contenant un type de données numérique, pour expérimenter un peu.

```
var maths = 5;
if( maths == 5 ) {
    console.log( "Je suis le chiffre cinq." );
}
```

Je suis le chiffre cinq.

Comme `if` évalue aveuglément le contenu des parenthèses qui suivent pour déterminer si l'expression est vraie, il n'est pas toujours nécessaire d'inclure une condition spécifique – on peut par exemple évaluer une valeur booléenne.

```
var foo = false;
if( foo ) {
    /* Le code placé ici ne s'exécutera jamais, à moins que
    vous ne changiez la valeur de 'foo' en 'true' */
}
```

else

L'instruction `else` sert à exécuter des lignes de code alternatives, si le contenu d'un `if` se révèle être `false`. L'instruction `else` suit l'accolade de fermeture de l'expression `if`, et elle est suivie d'une paire d'accolades contenant le code à exécuter au cas où le code `if` ne le serait pas. Nul besoin d'utiliser des parenthèses ici, car nous n'évaluons pas de nouvelles données – nous exécutons une action alternative, qui dépend des conditions de l'instruction `if`.

```
var maths = 2;
if( maths > 5 ) {
    console.log( "Supérieur à cinq." );
} else {
    console.log( "Inférieur ou égal à cinq." );
}
```

Inférieur ou égal à cinq.

else if

Il existe un raccourci permettant d'enchaîner plusieurs expressions `if` : `else if`. Si ce n'est pas nécessairement la façon la plus soignée d'effectuer des comparaisons complexes, cela n'en demeure pas moins une structure à connaître.

```
if( repas == "gravier" ) {
    console.log( "Berk." );
} else if ( repas == "burrito" ) {
    console.log( "Un burrito ? Excellent choix !" );
} else {
    console.log( "Ce n'est peut-être pas un burrito,
        mais au moins, ce n'est pas du gravier." );
}
```

Le script exécute une série de tests sur la variable `repas` : le premier `if` vérifie que le repas ne contient pas la valeur `"gravier"`, et une fois cette abomination culinaire écartée, nous pouvons vérifier s'il contient la valeur `"burrito"`, ou bien – avec la dernière instruction `else` – aucune des deux précédentes.

`else if` n'est pas une instruction JavaScript comme le sont `if` et `else` individuellement – `else if` est plutôt une sorte de solution syntaxique, un raccourci pour imbriquer plusieurs expressions `if/else`. Le code ci-dessus est structurellement identique à ce qui suit :

```
if( repas == "gravier" ) {
    console.log( "Berk." );
} else {
    if ( repas == "burrito" ) {
        console.log( "Un burrito ? Excellent choix !" );
    } else {
        console.log( "Ce n'est peut-être pas un burrito,
            mais au moins, ce n'est pas du gravier." );
    }
}
```

Qu'on utilise `else if` ou que l'on imbrique plusieurs expressions `if/else`, ce code n'est pas le plus simple à lire et à comprendre. Il existe de bien meilleures façons d'effectuer plusieurs comparaisons en même temps, et nous en aborderons quelques-unes dans un instant.

OPÉRATEURS DE COMPARAISON

Les expressions conditionnelles ne nous seraient pas d'une grande utilité si elles ne servaient qu'à tester si deux valeurs sont égales. Heureusement, et comme vous vous y attendez peut-être, nous pouvons faire bien plus de choses avec ces quelques expressions conditionnelles simples. Vous en avez vu un petit exemple précédemment, lorsque nous nous sommes servis d'un `if` pour déterminer si un identifiant avait une valeur de type `number` associée supérieure à cinq : les expressions conditionnelles peuvent servir à comparer toutes sortes de valeurs de toutes sortes de façons, et tout cela en remplaçant simplement le signe `==` que nous avons utilisé dans nos comparaisons jusqu'à présent.

Égalité

J'ai mentionné plusieurs fois qu'il fallait utiliser `==` pour les comparaisons, mais ce n'est pas strictement (et vous comprendrez bientôt ce piètre jeu de mots) vrai.

JavaScript nous offre deux approches différentes pour établir des comparaisons : le `==` que nous avons utilisé jusqu'à présent, et `===`, qui signifie « strictement égal ». Un double signe égal correspond à une comparaison « faible » entre deux valeurs, ce qui signifie qu'en saisissant `2 == "2"`; dans notre console de développement, on obtient le résultat *true*, alors que l'on compare un chiffre à une chaîne de caractères. JavaScript est assez intelligent pour comparer positivement deux types de données différents lorsque la comparaison est effectuée avec `==`, et deviner ce que l'on souhaite comparer.

`2 === "2"`; en revanche, renverra le résultat *false* – aucune conversion de type n'est effectuée en coulisses lorsqu'une comparaison est effectuée avec `===`. Les deux valeurs comparées doivent non seulement être égales, mais aussi être du même type : elles doivent être identiques.

Si vous trouvez que `==` a l'air un peu trop magique pour son propre bien, vous avez raison – les développeurs préfèrent largement utiliser `===` dans la mesure du possible, car il élimine toute ambiguïté pouvant résulter de la conversion automatique.

Truthy et falsy

Il y a une chose potentiellement utile que `==` permet de faire et pas `===` :

deviner quelles sont les valeurs *truthy* et les valeurs *falsy*.

Ces mots barbares signifient qu'en JavaScript, tout peut être converti en une valeur booléenne `true` ou `false` lorsqu'on utilise l'opérateur de comparaison non stricte.

N'ayez crainte, vous n'aurez pas besoin de tenir un tableau de valeurs *truthy* et *falsy*, car celles-ci suivent un raisonnement logique : « S'il y a quelque chose, *truthy* ; s'il n'y a rien, *falsy*. » Par exemple, `0` est une valeur *falsy* – tout comme `null`, `undefined`, `NaN` ou une chaîne de caractères vide (`""`). Tout le reste est *truthy* : une chaîne de caractères (non vide), un nombre (non nul), etc.

Les usages de ce principe ne vous paraîtront peut-être pas immédiatement évidents, mais imaginez une situation où vous écrivez une fonction qui produit une chaîne de caractères – comme celles que nous avons utilisées pour étudier la concaténation de chaînes :

```
function bienvenue( nom ) {  
    console.log( "Bienvenue, " + nom + " !" );  
}  
bienvenue( "Monsieur Muscle" );  
Bienvenue, Monsieur Muscle !
```

Comme nous l'avons vu, l'omission de l'argument contenant le nom de l'utilisateur ne produit pas une erreur, car JavaScript sait que la variable `nom` existe, mais la valeur `undefined` se retrouve convertie en une chaîne de caractères.

```
bienvenue();  
Bienvenue, undefined !
```

Ce comportement n'est pas particulièrement souhaitable ; personnellement, je n'apprécierais pas d'être appelé « undefined ». Heureusement, nous pouvons utiliser une expression `if/else` pour modifier légèrement le résultat :

```
function bienvenue( nom ) {  
    if( nom ) {  
        console.log( "Bienvenue, " + nom + " !" );  
    } else {  
        console.log( "Bienvenue, qui que vous soyez !" );  
    }  
}
```

Par défaut, JavaScript évalue le contenu de ces parenthèses en les

convertissant en valeurs booléennes – il recherche les valeurs *truthy* et *falsy*, et une chaîne de caractères non vide est une valeur *truthy*. Par conséquent, si nous essayons cette nouvelle fonction sans fournir de nom en guise d'argument, voilà ce qu'il se passe :

```
bienvenue( "Mat" );  
Bienvenue, Mat !  
bienvenue();  
Bienvenue, qui que vous soyez !
```

Ça marche !

Cette fonction n'est pas des plus commodes, cependant. Notre code serait mieux organisé et moins redondant si nous pouvions réduire ces deux instructions `console.log` en une seule. Pour quiconque se retrouvera à entretenir notre code après nous (et pour notre propre santé mentale), il est judicieux que nos scripts soient aussi sobres que possible. Vous verrez fréquemment ce concept désigné en anglais par l'acronyme DRY, pour *Don't repeat yourself* (ne vous répétez pas). Si vous devez changer quelque chose dans votre code par la suite, il sera préférable de n'avoir à le faire qu'à un seul endroit. Dans notre exemple, il pourrait simplement s'agir de changer le « Bienvenue » en « Salut », mais dans un script considérablement plus complexe, il serait impossible de tenir un inventaire mental de tous les endroits où vous devriez apporter des changements identiques. Un code DRY s'attache à suivre un cheminement simple.

Ainsi, dans notre fonction, au lieu d'avoir deux lignes qui produisent deux chaînes différentes dans la console, nous pouvons définir conditionnellement la chaîne elle-même et produire le résultat global à la fin.

```
function bienvenue( nom ) {  
  if( nom === undefined ) {  
    nom = "qui que vous soyez";  
  }  
  console.log( "Bienvenue, " + nom + "!" );  
}
```

Beaucoup plus succinct. Si `nom` ne contient aucune valeur, nous lui en donnons une : plutôt que de nous reposer sur la conversion *truthy/falsy* en sachant que nous avons potentiellement une variable `undefined`, nous vérifions spécifiquement ce point en premier. Puis, au moment où nous arrivons à la déclaration `console.log`, nous savons que la variable `nom` est définie.

Pour éviter que mon propre code ne devienne trop compliqué, l'une de mes méthodes préférées consiste à le décrire à voix haute. Précédemment, notre code suivait la logique suivante :

Si `nom` porte une valeur `truthy`, produire une chaîne contenant `nom` dans la console, mais si `nom` porte une valeur `falsy`, produire une chaîne alternative dans la console.

C'est aussi difficile à décrire à voix haute qu'à lire dans le code lui-même. Comparez cela à la description de notre nouvelle fonction :

Si la variable `nom` est `undefined`, la définir. Produire une chaîne contenant `nom` dans la console.

Voilà qui est bien mieux – et bravo, vous venez de réaliser votre première refactorisation en JavaScript.

Inégalité

`!` est-ce que l'on appelle un opérateur logique NON, ce qui signifie qu'il inverse la valeur qui suit immédiatement :

```
true;  
true  
false;  
false  
!true;  
false
```

Lorsque nous utilisons l'opérateur logique NON (`!`) devant un autre type de données (un nombre ou une chaîne de caractères, par exemple), celui-ci inverse la valeur *truthy/falsy* de cette donnée.

```
"chaîne";  
"string"  
!"chaîne";  
false  
0;  
0  
!0;  
true
```

>	Supérieur à
<	Inférieur à
>=	Supérieur ou égal à
<=	Inférieur ou égal à

FIG 3.1 : Un bref résumé des opérateurs relationnels.

De la même façon que `==` et `===` renvoient la valeur `true` si les deux valeurs comparées sont égales (strictement ou non), les opérateurs `!=` et `!==` renvoient une valeur `true` si les deux valeurs comparées ne sont pas égales. C'est un peu difficile à imaginer comme ça, mais c'est beaucoup plus logique dans le contexte d'une expression `if`.

```
var foo = 2;
if( foo != 5 ) {
    console.log( "'foo' n'est pas égal à cinq" );
}
```

Comme `==`, `!=` essaie de convertir les types de données comparés de sorte qu'ils soient comparables. Si l'on utilise `!=` pour comparer le chiffre `2` à la chaîne `"2"`, JavaScript considère que les deux sont égaux, et renvoie donc le résultat `false`.

```
2 != "3";
true
2 != "2";
false
```

Opérateurs relationnels

Les opérateurs relationnels sont un peu plus intuitifs que les opérateurs d'égalité ([FIG 3.1](#)).

Ils fonctionnent exactement comme on pourrait s'y attendre : pas de piège, pas d'opérateurs négatifs compliqués. On les utilise pour comparer une valeur numérique à une autre :

```
3 > 1;
true
3 < 1;
false
10 >= 5;
```

true

5 >= 5;

true

var nomVelo = "Bonneville";

if(nomVelo.length >= 10) {

console.log("Le nom de ce vélo comporte au moins
dix caractères.");

}

OPÉRATEURS LOGIQUES

Les expressions `if/else` peuvent faire beaucoup de travail en n'utilisant que ce que nous avons étudié jusqu'à présent, mais les opérateurs logiques permettent de développer un raisonnement encore plus complexe en enchaînant plusieurs comparaisons dans une seule expression. Vous avez déjà rencontré l'un de ces opérateurs logiques, à savoir l'opérateur NON (!) qui inverse toute valeur qui le suit, mais ! est l'opérateur le moins usité par rapport aux deux autres : les opérateurs OU (||) et ET (&&).

Les opérateurs || et && permettent d'évaluer plusieurs valeurs dans une même expression : plusieurs comparaisons séparées par || signifient que l'expression tout entière renverra *true* si une seule de ces expressions est vraie, tandis que les comparaisons séparées par && signifient que l'expression renverra *true* uniquement si toutes les expressions sont vraies. Là encore, c'est difficile à visualiser sans voir un exemple, alors revenons à notre console de développement :

```
5 < 2 || 10 > 2;
```

```
true
```

Il est évident que cinq n'est pas inférieur à deux – cette expression à elle seule ne renverrait jamais *true*. Cependant, dix est supérieur à deux – et comme nous utilisons un opérateur logique OU entre les deux comparaisons, l'expression entière renvoie la valeur *true*.

```
10 > 5 && "toast" === 2;
```

```
false
```

Dix est supérieur à cinq, bien sûr, et cette partie de l'expression est vraie. Mais la chaîne de caractères `toast` n'a clairement rien à voir avec le chiffre deux ; les deux valeurs ne sont pas égales, et elles ne sont certainement pas strictement égales. Comme nous utilisons l'opérateur logique ET entre ces deux expressions et que l'une d'entre elles renvoie *false*, l'expression tout entière renvoie *false*.

Grouper des expressions

Plusieurs expressions séparées par && et/ou || (vous l'avez ? « et/ou » ?) seront évaluées de gauche à droite. Dans l'expression suivante, JavaScript s'arrête à la première comparaison, sans même évaluer la seconde.

```
2 + 2 === 9 && "maChaine".length > 2;
```


false

Comme JavaScript voit une expression qui renvoie *false* suivie d'un opérateur logique ET, l'expression entière ne saurait absolument pas être vraie. La même règle s'applique à l'opérateur OU :

```
2 + 2 !== 9 || "maChaine".length > 2;
```

true

Comme la première expression renvoie *true* et qu'elle est suivie par un opérateur logique OU, JavaScript n'a aucun besoin de continuer à évaluer l'expression – celle-ci produit immédiatement le résultat *true*.

On peut modifier le comportement de cette évaluation en utilisant des parenthèses, ce qui nous ramène à nos cours d'algèbre. Nous allons commencer avec un ensemble de trois expressions qui renvoient *true* toutes ensembles, et nous utiliserons des booléens pour les voir exactement comme JavaScript les perçoit :

```
false && true || true;
```

true

La première chose que JavaScript fait ici est de regarder si `false && true` renvoie *true*, ce qui n'est pas le cas. `true && true` renverrait *true*, mais « false ET » signifie que l'expression est certaine de renvoyer *false*. Mais un opérateur logique OU se trouve à la suite – de sorte qu'après avoir évalué la première moitié de l'expression comme étant *false*, JavaScript évalue maintenant la deuxième moitié de l'expression comme étant `false || true`. Comme nous utilisons un opérateur logique OU et que l'une des valeurs est *true*, cette expression entière – lue de gauche à droite – renvoie *true*.

Si nous ajoutons une paire de parenthèses autour de la seconde partie de l'expression, cependant, nous changeons la façon dont celle-ci sera évaluée :

```
false && ( true || true );
```

false

Maintenant, le premier point évalué par JavaScript est toujours `false &&`, mais les parenthèses signifient que tout ce qui suit doit être évalué comme une seule expression – il n'y a donc pas trois éléments à évaluer, mais deux. Comme le côté gauche de l'opérateur logique ET renvoie *false*, l'évaluation s'arrête ici. « false ET » ne pourra jamais renvoyer *true*, donc avec une seule paire de parenthèses, nous avons changé cette expression en *false*.

Vous commencez à avoir la migraine ? Moi aussi, et je travaille là-dessus tous les jours. C'est pour cela que j'ai pris l'habitude d'utiliser des parenthèses pour clarifier la façon dont JavaScript évalue ces expressions complexes autant que pour la modifier. Maintenant que nous savons que JavaScript

évalue les expressions encadrées par des parenthèses comme des expressions uniques, cette première expression – celle qui renvoyait *true* – sera sans doute un peu plus simple à lire lorsqu'elle sera écrite ainsi :

```
( false && true ) || true;  
true
```

Songez à l'exemple de `else if` alambiqué que nous avons vu précédemment. Maintenant que nous pouvons effectuer des comparaisons plus avancées dans le cadre d'une seule expression `if`, nous pouvons considérablement en réduire la complexité, même lorsque nous lui adjoignons d'autres fonctionnalités – par exemple, des tacos :

```
var repas = "tacos";  
if( repas !== "gravier" && ( repas === "burrito" ||  
    repas === "tacos" ) ) {  
    console.log( "Délicieux." );  
}
```

Et pour faire bonne mesure, transformons cela en une fonction :

```
function verifRepas( repas ) {  
    if( repas !== "gravier" && ( repas === "burrito" ||  
        repas === "tacos" ) ) {  
        console.log( "Délicieux." );  
    }  
}  
verifRepas( "Tacos" );  
undefined
```

Quelque chose a vraiment mal tourné ! Nous avons fait passer une chaîne de caractères à la fonction comme prévu, mais nous avons fait une petite erreur : nous avons mis un *T* majuscule. Comme JavaScript est sensible à la casse, `Tacos` n'est pas égal à `tacos` – la valeur attendue par notre script.

Nous pourrions établir une règle spécifiant que cette fonction devra toujours recevoir une valeur en minuscules, mais c'est une chose de plus à documenter – ou pire, une chose de plus à « garder à l'esprit ». Une meilleure approche consisterait à prévoir l'utilisation de valeurs en majuscules et en minuscules, en demandant à JavaScript de normaliser le tout pour nous.

Vous vous souvenez peut-être de ce que nous avons vu au chapitre précédent : toute chaîne que l'on définit s'accompagne d'un ensemble de propriétés intégrées. Par exemple, `.length` vous donne le nombre de caractères de la

chaîne. Nous allons utiliser ici l'une des méthodes natives pour transformer une chaîne afin d'être sûr de comparer ce qui est comparable : `.toLowerCase()`. Comme son nom l'indique, cette méthode permet de renvoyer la valeur d'une chaîne en minuscules :

```
"CECI EST UNE CHAÎNE".toLowerCase();  
ceci est une chaîne
```

Si elle renvoie bien la valeur d'une chaîne en minuscules, il est important de savoir que cette méthode – et d'autres comme elle – ne modifie pas la chaîne de caractères. Si une chaîne est stockée dans une variable et qu'on lui applique la méthode `.toLowerCase()`, la variable reste inchangée.

```
var foo = "Une Chaîne";  
undefined  
foo;  
"Une Chaîne"  
foo.toLowerCase();  
"une chaîne"  
foo;  
"Une Chaîne"
```

Il existe donc plusieurs moyens de gérer la comparaison dans notre fonction. La première consiste à appliquer `.toLowerCase()` chaque fois que nous appelons la variable `repas` dans la fonction :

```
function verifRepas( repas ) {  
  if( repas.toLowerCase() !== "gravier" && (  
    repas.toLowerCase() === "burrito" ||  
    repas.toLowerCase() === "tacos" ) ) {  
    console.log( "Délicieux." );  
  }  
}  
verifRepas( "Tacos" );  
Délicieux.
```

Ça marche, certes, mais ce n'est pas du code très DRY. Nous pouvons faire mieux que ça. Nous allons plutôt utiliser `.toLowerCase()` une seule fois, au début de la fonction, avant la comparaison, pour affecter temporairement à la variable `repas` la version en minuscules renvoyée par `.toLowerCase()`.

```
function verifRepas( repas ) {  
  repas = repas.toLowerCase();
```

```
    if( repas !== "gravier" && ( repas === "burrito" ||  
        repas === "tacos" ) ) {  
        console.log( "Délicieux." );  
    }  
}  
verifRepas( "TACOS" );  
Délicieux.
```

Inutile d'employer `var` pour affecter une valeur à `repas`, car nous utilisons `repas` comme argument, ce qui signifie que nous l'avons déjà défini comme une variable locale pour la fonction.

switch

Une instruction `switch` fonctionne essentiellement comme ces assemblages complexes d'`else if` que nous avons vus, mais effectue les mêmes comparaisons d'une manière plus compacte et efficace. La syntaxe est un peu différente des `if` que nous avons rencontrés jusqu'à présent :

```
var leNumero = 5;
switch( leNumero ) {
  case 1:
    console.log( "Numéro un." );
    break;
  case 2:
    console.log( "Numéro deux." );
    break;
  case 3:
  case 4:
    console.log( "Numéro trois ou quatre." );
    break;
  case 5:
    console.log( "Numéro cinq." );
}
```

Il se passe beaucoup de choses ici, alors étudions ce bout de code ligne par ligne. Avant toute chose, souvenez-vous que JavaScript n'a que faire de l'espacement – toutes ces indentations sont là pour une question de lisibilité, mais elles ne sont pas strictement nécessaires.

La première ligne ne devrait plus vous être inconnue à ce stade : nous définissons une variable avec l'identifiant `leNumero` et lui attribuons la valeur numérique `5` – et comme nous ne faisons que bricoler, nous ne nous soucierons pas du fait que `leNumero` n'est pas un identifiant très descriptif.

La deuxième ligne ressemble un peu aux expressions `if` que vous avez appris à connaître et à aimer (je l'espère) : l'instruction `switch` est suivie d'une paire de parenthèses, puis d'une paire d'accolades. `switch` diffère de `if` dans le sens où nous n'effectuons pas de comparaison entre les parenthèses : nous faisons simplement passer les informations à comparer, comme nous faisons passer une chaîne de caractères à notre fonction il y a quelques minutes. Tout ce que dit l'instruction `switch (leNumero) { }`, c'est que la variable

`leNumero` est la valeur que nous souhaitons comparer à l'intérieur de l'expression `switch` – vous comprendrez mieux dans un moment.

C'est à la troisième ligne (`case 1:`) que la comparaison s'opère. L'instruction `case` est suivie de la valeur qui sera comparée à celle que nous avons saisie entre parenthèses après l'instruction `switch`, puis d'un deux-points. En d'autres termes, la ligne `case 1:` dit : « si `leNumero` est égal à un, faire ce qui suit ». Toutes les comparaisons effectuées dans un `switch` sont strictes – `case "1":` ne correspondrait pas, `"1"` étant une chaîne de caractères et non une valeur numérique.

```
switch( "1" ) {  
  case 1:  
    console.log( "Numéro un." );  
    break;  
  case "1":  
    console.log( "Chaîne de caractères '1'" );  
}
```

Chaîne de caractères '1'

L'instruction `break` indique ensuite « on a trouvé une correspondance, arrêtez la comparaison ». On ne souhaite pas toujours arrêter immédiatement la comparaison ; quelques lignes plus loin, vous verrez qu'on vérifie `case 3` et `case 4` ; ensuite, si l'une de ces deux comparaisons correspond, on écrit avec `console.log` que la valeur est de trois ou quatre, puis on place le `break`. Si cela marche, c'est parce qu'une instruction `case` correspondante placée dans un `switch`, à proprement parler, dit à JavaScript : « Exécute chaque petite ligne de code qui suit le `case` correspondant jusqu'à une instruction `break` (ou l'accolade de fin de l'expression `switch`). »

Pour illustrer ce comportement, admettons que l'on veuille créer une fonction (pour une raison quelconque) qui accepte les jours de la semaine en valeur numérique (1-7) et renvoie le nom des jours qui se sont écoulés depuis le début de la semaine.

```
function joursEcoulesCetteSemaine( jourNumerique ) {  
  console.log( "Les jours suivants se sont déjà écoulés  
    depuis le début de la semaine :" );  
  switch( jourNumerique ) {  
    case 7:  
      console.log( "Samedi" );  
    case 6:
```

```

        console.log( "Vendredi" );
    case 5:
        console.log( "Jeudi" );
    case 4:
        console.log( "Mercredi" );
    case 3:
        console.log( "Mardi" );
    case 2:
        console.log( "Lundi" );
    case 1:
        console.log( "Dimanche" );
        break;
    default:
        console.log( "Euh, ce n'est pas un
            jour de la semaine." );
    }
}

joursEcoulesCetteSemaine( 3 ); /* Nous sommes le
    troisième jour de la semaine. */
Les jours suivants se sont déjà écoulés depuis le début de la
semaine :
Mardi
Lundi
Dimanche

```

Ce n'est pas la fonction la plus utile, mais vous saisissez l'idée : lorsqu'on entre la valeur `3`, toutes les expressions `console.log` entre la ligne `case 3:` et l'instruction `break` sont exécutées, et on obtient une liste de jours.

Vous remarquerez également une nouveauté dans cette expression `switch` : une instruction `default`.

L'instruction `default` est comme le `else` d'un `switch`, si cette phrase veut dire quelque chose : dans le cas où aucune des valeurs `case` ne renverrait `true`, le code qui suit `default` sera exécuté.

```

joursEcoulesCetteSemaine( 75 );
Les jours suivants se sont déjà écoulés depuis le début de la
semaine :
Euh, ce n'est pas un jour de la semaine.

```

Comme nous écrivons le code `default` en dernier, nous n'avons pas besoin de placer un `break` à la suite – mais il en faut un avant le `default`, sans quoi cette erreur apparaîtra avec le reste de notre liste. Toutefois, si vous préférez, la fonction ci-dessus peut être écrite avec l'instruction `default` en premier, suivie d'un `break`.

```
function joursEcoulesCetteSemaine( jourNumerique ) {  
    console.log( "Les jours suivants se sont déjà écoulés  
        depuis le début de la semaine :" );  
    switch( jourNumerique ) {  
        default:  
            console.log( "Euh, ce n'est pas un  
                jour de la semaine." );  
            break;  
        case 7:  
            console.log( "Samedi" );  
        case 6:  
            console.log( "Vendredi" );  
        case 5:  
            console.log( "Jeudi" );  
        case 4:  
            console.log( "Mercredi" );  
        case 3:  
            console.log( "Mardi" );  
        case 2:  
            console.log( "Lundi" );  
        case 1:  
            console.log( "Dimanche" );  
    }  
}  
  
joursEcoulesCetteSemaine( 5 ); /* Nous sommes le  
    cinquième jour de la semaine. */
```

Les jours suivants se sont déjà écoulés depuis le début de la semaine :

Jeudi

Mercredi

Mardi

Lundi

Dimanche

`switch` peut sembler curieux, mais dans certaines situations, il est particulièrement utile d'effectuer une série de comparaisons avec un seul objet. Par exemple, imaginez un script qui accepte la saisie au clavier et l'utilise pour déplacer un sprite – beaucoup de jeux vous demanderont d'utiliser les touches fléchées ou les touches *A* et *D* pour vous déplacer de gauche à droite. En JavaScript, les pressions de touches sont représentées par un objet d'événement (que nous aborderons dans un petit moment) avec une propriété contenant une valeur numérique qui correspond à la touche pressée.

```
function bougerJoueur( codeTouche ) {  
  switch( codeTouche ) {  
    case 65: // Code pour la touche A  
    case 37: // Code pour la flèche de gauche  
      bougerGauche ();  
      break;  
    case 68: // Code pour la touche D  
    case 39: // Code pour la flèche de droite  
      bougerDroite();  
  }  
}
```

Vous pourriez écrire ce même code sous la forme d'une série d'expressions `if`, mais pour ajouter de nouveaux contrôles au fil du temps, il faudrait enchaîner un `if` après l'autre, chacun effectuant une comparaison avec le même objet `codeTouche`. `switch` s'avère être une méthode bien plus flexible (et plus DRY) de procéder.

J'AI LE TOURNIS

Nous avons vu plusieurs méthodes permettant d'exprimer des logiques extrêmement complexes, simplement en utilisant les humbles instructions `if` et `switch`. Cela fait beaucoup de choses à retenir, mais au risque de me répéter : nul besoin de tout apprendre par cœur. Vous pouvez simplement tourner la page en sachant qu'il existe une façon d'exprimer la logique conditionnelle dont vous pourriez avoir besoin – et si vous ne parvenez pas à vous souvenir de la syntaxe exacte, eh bien, ce chapitre ne disparaîtra pas de sitôt.

Les expressions conditionnelles nous permettent d'exécuter du code de façon sélective, mais ce n'est pas tout. La programmation demande régulièrement d'exécuter un ensemble de tâches de manière répétée, par exemple pour parcourir tous les éléments d'un certain type sur une page et vérifier un attribut commun par rapport à une valeur attendue, ou parcourir tous les éléments d'un tableau et vérifier leur valeur par rapport à une expression conditionnelle. Pour tout cela et plus encore, nous allons apprendre à nous servir des boucles.

4

LES BOUCLES

LES BOUCLES NOUS PERMETTENT de répéter des lignes de code jusqu'à ce que certaines conditions soient remplies. C'est là encore un concept simple en surface, mais qui nous permettra de faire une quantité de travail surprenante.

for

Vous utiliserez des boucles `for` dans les situations où vous voudrez exécuter une boucle un nombre de fois, ou d'itérations, connu. (Par « connu », je n'entends pas que nous, pauvres mortels, sachions nécessairement à l'avance combien de fois la boucle sera exécutée ; je veux simplement dire que nous répéterons la boucle un nombre de fois donné.)

La syntaxe d'une boucle est un peu délicate, car on y donne beaucoup d'informations en bien peu de caractères. À ce stade, vous pouvez déjà imaginer quelles seront les composants de base : une instruction `for`, suivie d'une paire de parenthèses, suivie d'une paire d'accolades contenant les instructions que notre boucle devra répéter un certain nombre de fois.

Mais la syntaxe placée entre parenthèses est différente de tout ce que nous avons vu jusqu'à présent. Une boucle `for` accepte trois expressions : l'initialisation, la condition et l'expression finale, toutes séparées par des points-virgules.

```
for( var i = 0; i < 3; i++ ) {  
    console.log( "Cette boucle s'exécutera trois fois." );  
}
```

(3) Cette boucle s'exécutera trois fois.

L'initialisation sert pratiquement toujours à une seule chose : initialiser une variable qui servira de compteur. Celle-ci est initialisée comme n'importe quelle autre variable, à l'aide d'une instruction `var`, d'un identifiant et d'une valeur affectée. Vous verrez un grand nombre de ces variables « compteurs » porter l'identifiant `i`, pour « itération ». Il est vrai que cela va à l'encontre de la règle qui interdit de donner un nom d'un seul caractère à un identifiant, mais c'est une convention largement établie. Comme JavaScript commence son indexation à partir de zéro, il est judicieux de toujours partir de zéro nous aussi ; ainsi, nous ne prendrons pas la mauvaise habitude de compter à partir de un ou d'avoir des compteurs décalés dans nos scripts.

La condition sert à définir le point auquel la boucle s'arrête. Ainsi, nous définissons `i` comme partant de zéro, et nous voulons que la boucle s'exécute tant que `i` est inférieur à trois.

L'expression finale est l'instruction à exécuter à la fin de chaque itération de la boucle – c'est là que nous incrémentons la variable `i` en y ajoutant un. Si vous vous rappelez des opérateurs mathématiques que nous avons vus plus tôt, vous vous souviendrez que la syntaxe `++` incrémente une valeur de un :

`i++` comme expression finale signifie « augmenter `i` de un à chaque fois que la boucle se termine ».

En bon français, la syntaxe de `for` ci-dessus dit en substance : « Initialiser la variable `i` à zéro. Exécuter le code qui suit uniquement si `i` est inférieur à trois, puis ajouter un à `i` après chaque itération. »

L'un des usages les plus courants d'une boucle `for` consiste à parcourir chaque élément d'un tableau. Comme nous l'avons vu au chapitre précédent, les tableaux possèdent une propriété permettant de déterminer combien d'éléments ils contiennent – la propriété `.length` – donc nous travaillerons toujours avec une quantité connue. Si nous utilisons la longueur du tableau dans la condition, nous obtiendrons une boucle qui se répète autant de fois qu'il y a d'éléments dans notre tableau :

```
var tableauBoucle = ["premier", "deuxième", "troisième"];
for( var i = 0; i < tableauBoucle.length; i++ ) {
    console.log( "Boucle." );
}
```

(3) *Boucle.*

Et si nous ajoutons un élément à notre tableau test, le nombre d'itérations change en conséquence :

```
var tableauBoucle = ["premier", "deuxième", "troisième", 4];
for( var i = 0; i < tableauBoucle.length; i++ ) {
    console.log( "Boucle." );
}
```

(4) *Boucle.*

Super ! Nous pouvons construire une fonction qui exécute du code autant de fois que nous avons d'éléments dans un tableau. Je sais que ce n'est pas particulièrement utile ni enthousiasmant à première vue, mais ce qui est génial (je sais, je ne sors pas beaucoup), c'est que `i` est une bonne vieille variable de type numérique que nous pouvons utiliser à chaque itération de la boucle, et comme nous comptons à partir de zéro, elle contient une valeur qui correspond à l'indice de chaque élément de notre tableau. Une boucle `for` nous permet ainsi de parcourir toutes les données d'un tableau d'un seul coup :

```
var noms = [ "Ed", "Al" ];
for( var i = 0; i < noms.length; i++ ) {
    var nom = noms[ i ];
    console.log( "Salut, " + nom + " !" );
}
```

```
}  
Salut, Ed !  
Salut, Al !
```

Ainsi, nous pouvons parcourir le tableau `noms`, et en utilisant `i` comme indice, accéder à chaque élément du tableau.

Nous n'avons pas besoin d'initialiser une nouvelle variable `noms` ; nous aurions pu utiliser `noms[i]` dans `console.log`, et tout aurait fonctionné de la même manière. Stocker les données d'un tableau dans une variable à chaque itération est une bonne idée si vous êtes susceptible d'accéder à ces données plusieurs fois au cours de votre boucle, ne serait-ce que pour des raisons pratiques.

for/in

Les boucles `for/in` démarrent de la même façon que les boucles `for` ci-dessus, avec une instruction `for`, une paire de parenthèses et une paire d'accolades contenant le code à répéter. De la même façon, on utilise la syntaxe `for/in` pour parcourir plusieurs éléments, mais pas nécessairement comme avec un tableau. `for/in` sert à parcourir les propriétés d'un objet dans un ordre arbitraire et non séquentiel.

Au lieu d'une initialisation, d'une condition et d'une expression finale, une boucle `for/in` commence par l'initialisation d'une variable correspondant aux clés de notre objet, suivie du mot-clé `in` et de l'objet à parcourir :

```
var objetNom = {  
  "prénom": "Mat",  
  "nom de famille": "Marquis"  
};  
for( var nom in objetNom ) {  
  console.log( "Boucle." );  
}  
(2) Boucle.
```

Au lieu de parcourir un tableau avec une boucle `for` et une variable `i` bien pratique à notre disposition, il nous faut maintenant un peu plus réfléchir pour comprendre quelles chaînes sont utilisées comme clés dans notre objet. Ces clés sont affectées à la variable `nom` initialisée entre parenthèses.

```
var objetNom = {  
  "prénom": "Mat",
```

```
    "nom de famille": "Marquis"
};
for( var nom in objetNom ) {
    console.log( nom );
}
prénom
nom de famille
```

Pas très utile à première vue, mais de même qu'une boucle `for` ordinaire nous donnait un type de données numérique servant à accéder à nos données à chaque itération, `for/in` nous donne la chaîne dont nous avons besoin pour accéder aux données d'un objet :

```
var nomComplet = {
    "prenom": "Mat",
    "nom de famille": "Marquis"
};
for( var nom in nomComplet ) {
    console.log( nom + " : " + nomComplet[ nom ] );
}
prénom : Mat
nom de famille : Marquis
```

Vous remarquerez que nous utilisons la notation à crochets au lieu de la notation à point ; en fait, nous y sommes obligés. Si l'on tente d'accéder à `nomComplet.nom`, c'est exactement à cela qu'on accède : `nomComplet.nom`, une propriété avec la clé `nom` à l'intérieur de `nomComplet`, et non une propriété avec une clé correspondant à la chaîne que `nom` contient.

Vous vous demandez peut-être : « Mais si à peu près tout est un objet, est-ce que cela signifie que nous pouvons utiliser `for/in` pour parcourir un tableau par itération ? » De fait, c'est possible. Les tableaux se comportent comme n'importe quel autre objet en utilisant `for/in`, mais `for/in` n'est pas aussi adapté aux tableaux qu'une boucle `for` ordinaire. D'une part, nous ne pouvons pas garantir que `for/in` parcourra le tableau dans l'ordre séquentiel – ce qui peut être acceptable, selon ce que vous cherchez à faire.

Le plus gros problème, c'est que `for/in` présente une particularité qu'on ne retrouve pas dans la boucle `for` : comme pratiquement tout est un objet – et que vous pouvez ajouter des propriétés à n'importe quel objet – `for/in` peut se retrouver à parcourir des propriétés dont nous ne voulions pas qu'il connaisse l'existence.

J'ai mentionné précédemment que tout en JavaScript comportait des méthodes et des propriétés « intégrées » – qu'une chaîne, même si nous ne la définissons que comme une poignée de caractères, s'accompagnera toujours de propriétés telles que `.length` et de méthodes telles que `.toLowerCase()`.

Ces méthodes et ces propriétés ne sont pas complètement enfouies dans les sombres recoins de JavaScript. Nous pouvons voir, et même changer les méthodes et les propriétés attachées aux types de données, tableaux, objets, etc.

L'héritage prototypal

La plupart des objets ont une propriété interne nommée `prototype` qui contient ces propriétés intégrées, mais il est un peu étrange d'y accéder nous-mêmes. Lorsque nous accédons à une propriété d'un objet, même si nous l'avons créée nous-mêmes, JavaScript vérifie d'abord si c'est nous qui l'avons définie. Sinon, il cherche cette clé en tant que propriété de `prototype` du constructeur de cet objet – une sorte de template global que JavaScript suit lorsqu'il doit traiter un certain type d'objet. Toutes les chaînes, par exemple, héritent des propriétés et des méthodes `prototype` du constructeur `String`. Dans la plupart des navigateurs, vous pouvez saisir `String.prototype`; dans la console pour découvrir toutes ces propriétés intégrées.

Si votre premier réflexe est d'avoir peur que JavaScript nous permette de détourner des méthodes intégrées, vous avez totalement raison. La méthode `toString`, par exemple, est un moyen bien tranché de convertir n'importe quel objet en une chaîne, mais avec quelques lignes de code, nous pouvons l'écraser et faire en sorte que cette méthode se plie à notre bon plaisir.

```
var monObjet = {};  
var autreObjet = {};  
monObjet.toString();  
    "[object Object]"  
monObjet.toString = function() {  
    console.log( "Je crois que j'ai cassé JavaScript." );  
};  
monObjet.toString();  
Je crois que j'ai cassé JavaScript.  
autreObjet.toString();  
    "[object Object]"
```

C'est un pouvoir démesuré, mais attendez, ce n'est pas tout. Nous pouvons

accéder au `prototype` d'un constructeur à partir de tout objet de ce type en utilisant la propriété `__proto__` – une référence au `prototype` de tous les objets du même type. Elle n'est pas encore disponible dans tous les navigateurs, mais ce n'est pas grave, il vaut sans doute mieux éviter d'y toucher de toute façon. `__proto__` permet d'ajouter, de supprimer et de modifier complètement l'action des propriétés JavaScript intégrées sur un objet donné – et ce faisant, de modifier l'ensemble des propriétés et des méthodes qui sont intégrées dans tous les objets associés.

```
var monObjet = {};  
var autreObjet = {};  
monObjet.toString();  
"[object Object]"  
monObjet.__proto__.toString = function() {  
    console.log( "Je crois que j'ai VRAIMENT cassé  
    JavaScript." );  
};  
monObjet.toString();  
Je crois que j'ai VRAIMENT cassé JavaScript.  
autreObjet.toString(); // Oups...  
Je crois que j'ai VRAIMENT cassé JavaScript.
```

Nous avons ici réussi à changer la méthode `toString`, non seulement pour `monObjet`, mais pour tous les objets, en la modifiant au niveau de `prototype`. Quand notre script se limite à quelques lignes, la perspective n'est peut-être pas si terrifiante, mais en modifiant ainsi le fonctionnement de JavaScript, nous courons le risque de faire boguer un site tout entier.

Heureusement, vous ne verrez pas ça très souvent. Lorsque quelqu'un a fait suffisamment de progrès en programmation pour ressentir le besoin de toucher à la propriété `prototype`, il a généralement appris à ne pas le faire. Cependant, vous verrez parfois des ajouts à `prototype` – des méthodes et des fonctions ajoutées pour être mises à disposition des objets du même type.

On peut ajouter des méthodes et des propriétés pour tous les objets d'un certain type en modifiant directement la propriété `prototype` d'un constructeur, mais il ne faut pas modifier les propriétés qui sont déjà définies. Vous pouvez ainsi faire des ajouts à `String.prototype` comme vous ajouteriez des propriétés à un objet que vous avez créé vous-même.

Ajouter des méthodes à `prototype` n'est pas une idée brillante – surtout pour les besoins des boucles `for/in`. Mais aux fins de la discussion, admettons que

vous avez besoin d'examiner fréquemment des objets à la recherche d'une clé portant l'identifiant `nom`. En théorie, nous pourrions simplement ajouter une méthode à tous les objets, en ajoutant une nouvelle méthode à la propriété `prototype` du constructeur `Object`. Nous n'aurons pas besoin d'utiliser `__proto__` ici, car nous ne cherchons pas à changer `Object.prototype` par le biais d'un objet :

```
var premierObjet = {
  "foo" : false
};
undefined
var deuxiemeObjet = {
  "nom" : "Hawkeye",
  "lieu" : "Maine"
};
undefined
Object.prototype.contientUnNom = function() {
  var resultat = false;
  for( var cle in this ) {
    if( cle === "nom" ) {
      resultat = true;
    }
  }
  return resultat;
}
function Object.contientUnNom()
premierObjet.contientUnNom();
false
deuxiemeObjet.contientUnNom();
true
```

Une fois que vous aurez ajouté une propriété ou une méthode au `prototype` d'un objet, celle-ci sera disponible pour toutes les instances de ce type de données.

Une petite précision : n'apprenez pas ce code par cœur, car c'est un moyen excessivement compliqué de vérifier si un objet contient la clé `nom`. Cela marche plutôt bien, mais outre le fait que c'est un moyen étrange de gérer une tâche simple, cela produit un effet secondaire indésirable : les propriétés

intégrées d'un objet ne sont pas dénombrables, ce qui signifie qu'elles ne s'affichent pas quand on parcourt les propriétés d'un objet avec une boucle `for/in`. Lorsqu'on ajoute de nouvelles propriétés à `prototype`, en revanche, celles-ci sont dénombrables :

```
Object.prototype.contientUnNom = function() {
    var resultat = false;
    for( var cle in this ) {
        if( cle === "nom" ) {
            resultat = true;
        }
    }
    return resultat;
}

function Object.contientUnNom()
var nouvelObjet = { "nom": "BJ" };
for( var cle in nouvelObjet ) {
    console.log( cle );
}
nom
contientUnNom
```

Ainsi, à chaque fois que nous exécuterons une boucle `for/in`, notre méthode `contientUnNom` s'affichera, ce qui n'est certainement pas idéal et constitue probablement une raison suffisante pour laisser `prototype` tranquille. À chaque fois que nous bricolons `prototype`, nous apportons des changements globaux à la mécanique interne de JavaScript – en ajoutant une méthode ou une propriété, les boucles `for/in` de tous les scripts de notre page risquent de se comporter de manière inattendue.

Nous devons donc examiner le problème par l'autre bout de la lorgnette – qu'advient-il de nos boucles `for/in` lorsque quelqu'un d'autre commence à trafiquer `prototype` ? Une page peut contenir des scripts écrits par plusieurs développeurs, des scripts tiers provenant de sources externes, etc., et nous n'avons aucune certitude que nos boucles `for/in` ne seront pas affectées par des propriétés dénombrables inattendues.

hasOwnProperty

La raison de ce voyage périlleux à travers `prototype` était double : tout

d'abord, je voulais vous faire découvrir `prototype`, puisque vous étiez dans le coin et tout ça. Ensuite, je voulais vous présenter la méthode `hasOwnProperty`, que nous pouvons utiliser pour protéger nos boucles `for/in` contre les propriétés dénombrables inattendues sur `prototype` :

```
Object.prototype.contientUnNom = function() {
    var resultat = false;
    for( var cle in this ) {
        if( cle === "nom" ) {
            resultat = true;
        }
    }
    return resultat;
}
function Object.contientUnNom()
var objetMystere = {
    "nom" : "Frank"
};
objetMystere.hasOwnProperty( "nom" );
true
objetMystere.hasOwnProperty( "contientUnNom" );
false
```

Il se trouve justement que `hasOwnProperty` peut permettre de faire ce que nous essayions d'accomplir en jouant avec `prototype` : déterminer si un objet que nous avons créé contient une certaine propriété. Mais plus important encore (du moins, pour les besoins de nos boucles), `hasOwnProperty` ne s'applique pas aux propriétés héritées de `prototype`. Nous pouvons l'utiliser pour protéger nos boucles `for/in` contre les magouilles de `prototype` malavisées.

```
Object.prototype.methodePrototypePersonnalisee =
function() {
    console.log( "Rebonjour." );
};
function Object.methodePrototypePersonnalisee()
var swamp = {
    "bunk1" : "Hawkeye",
    "bunk2" : "BJ",
```

```

    "bunk3" : "Frank"
};
undefined
for( var bunk in swamp ) {
    console.log( swamp[ bunk ] );
}
Hawkeye
BJ
Frank
function Object.contientUnNom()
for( var bunk in swamp ) {
    if( swamp.hasOwnProperty( bunk ) ) {
        console.log( swamp[ bunk ] );
    }
}
Hawkeye
BJ
Frank

```

Une fois `hasOwnProperty` en place, aucun ajout à `prototype` ne pourra modifier les résultats que nous attendons de nos boucles `for/in`. Après tout, nous ne pourrons pas toujours garantir que nous contrôlions chaque ligne de code qui se trouve dans nos sites web, ni savoir avec certitude si un autre développeur a décidé de modifier le `prototype` d'un constructeur.

while

Après cette brève excursion au pays de `prototype`, la syntaxe des boucles `while` devrait être un changement de ton bienvenu. Comme toutes les autres, elle commence par une instruction (`while`) suivie d'une paire de parenthèses et d'accolades. La seule chose que nous mettrons entre les parenthèses d'une boucle `while` est une condition – et comme l'implique l'instruction, la boucle continuera de s'exécuter aussi longtemps que cette condition renvoie `true`.

```
var i = 0;
while( i < 3 ) {
    console.log( "Boucle." );
    i++;
}
(3) Boucle.
```

Le code ci-dessus est simplement une autre manière d'écrire notre première boucle `for`. Au lieu de placer l'initialisation, la condition et l'expression finale entre les parenthèses, on crée la variable du compteur avant la boucle et on incrémente le compteur à l'intérieur de la boucle.

Ce n'est toutefois pas un cas d'utilisation courant pour une boucle `while` : on pourrait écrire tout cela de manière bien plus concise en utilisant `for`, après tout. On utilise `while` lorsque l'on n'a aucun moyen de mesurer le nombre d'itérations nécessaire et qu'il faut que la boucle continue à s'exécuter jusqu'à ce qu'une certaine condition soit satisfaite. Par exemple, le petit bout de code suivant continuera à générer un nombre aléatoire entre zéro et neuf et ne s'arrêtera que lorsque ce nombre aléatoire sera trois :

```
var nbAleatoire = Math.floor( Math.random() * 10 );
while( nbAleatoire !== 3 ){
    console.log( "Non, pas " + nbAleatoire );
    nbAleatoire = Math.floor( Math.random() * 10 );
}
console.log( "Trouvé !" );
Non, pas 5
Non, pas 9
Non, pas 2
Trouvé !
```

Si l'on exécute à nouveau ce code, la boucle `while` s'exécutera autant de fois

que nécessaire avant de passer à l'instruction `console.log` qui suit. Il se peut aussi que la méthode `random` produise un 3 dès le premier essai, auquel cas la boucle ne s'exécutera jamais.

do/while

Les boucles `do/while` poursuivent essentiellement le même objectif que les boucles `while` : répéter une boucle autant de fois que nécessaire jusqu'à ce qu'une condition donnée renvoie `true`. La syntaxe est légèrement différente – en fait, par rapport à la logique conditionnelle que vous avez vue jusqu'à présent, les boucles `do/while` peuvent sembler un peu arriérées. Nous commençons par une instruction `do`, immédiatement suivie d'une paire d'accolades – pas de parenthèses, pas de conditions – contenant le code que nous souhaitons répéter. Après les accolades, nous plaçons l'instruction `while` et les parenthèses qui contiennent la condition ; tant que cette condition renverra `true`, la boucle continuera de s'exécuter.

```
var i = 0;
do {
  console.log( "Boucle." );
  i++;
} while (i < 3);
(3) Boucle.
```

Il n'y a vraiment qu'une différence entre une boucle `do/while` et une bonne vieille boucle `while` : le contenu d'une boucle `while` est susceptible de ne jamais être exécuté (comme dans notre exemple de nombre aléatoire ci-dessus), mais le code d'une boucle `do/while` s'exécutera toujours au moins une fois.

Au lieu d'évaluer la condition avant de décider s'il faut exécuter le code ou non, comme le font toutes les autres boucles, `do/while` exécute le code, puis arrête la boucle une fois que la condition est satisfaite. Si nous réécrivions notre petit programme de nombre aléatoire en utilisant `do/while`, le code s'exécuterait une fois, même si la condition était satisfaite dès le départ :

```
do {
  var nbAleatoire = Math.floor( Math.random() * 10 );
  console.log( "Est-ce que c'est... " + nbAleatoire + "
?" );
} while( nbAleatoire !== 3 );
console.log( "Trouvé !" );
```

Est-ce que c'est... 7 ?
Est-ce que c'est... 9 ?
Est-ce que c'est... 6 ?
Est-ce que c'est... 3 ?
Trouvé !

Dans ce cas, nous n'avons pas besoin de générer un nombre aléatoire avant la boucle et d'en générer un à nouveau à chaque itération : comme le code à l'intérieur des accolades de `do` sera toujours exécuté avant que la condition ne soit évaluée, nous pouvons générer le nombre aléatoire pour la première exécution et le générer à nouveau à chaque itération consécutive de cette seule ligne. Nous affichons ensuite chaque « tentative » dans la console – et comme le code est exécuté avant que la condition ne soit vérifiée, cela inclura le nombre correspondant. Une fois la condition satisfaite, la boucle s'arrête et on passe à la suite.

continue ET break

La syntaxe de toutes les boucles gère à la fois l'itération et la terminaison – elles acceptent toutes une forme de condition pour arrêter la répétition. Si nous avons besoin d'un contrôle plus précis, nous pouvons guider nos boucles à l'aide des instructions `continue` et `break`.

Pour jouer un peu avec ces instructions, commençons par coder une boucle `for` qui compte de zéro à quatre :

```
for( var i = 0; i < 5; i++ ) {  
    console.log( i );  
}  
0  
1  
2  
3  
4
```

`continue` nous permet de passer directement à la prochaine itération d'une boucle sans exécuter le code placé après l'instruction `continue`.

```
for( var i = 0; i < 5; i++ ) {  
    if( i === 2 ) {  
        continue;  
    }  
    console.log( i );  
}  
0  
1  
3  
4
```

Cette boucle saute le troisième `console.log` en utilisant `continue` lorsque `i` est égal à deux – ce qui saute la troisième itération, puisque, vous vous en souvenez, nous commençons à compter à partir de zéro. Le numéro 2 n'apparaîtra jamais dans notre console.

`break` – une instruction profondément satisfaisante à taper après plusieurs heures de programmation – arrête non seulement l'itération actuelle d'une boucle, mais aussi la boucle tout entière :

```
for( var i = 0; i < 5; i++ ) {  
    if( i === 2 ) {  
        break;  
    }  
    console.log( i );  
}  
0  
1
```

Lorsqu'il rencontre l'instruction `break`, JavaScript réagit de la même façon que lorsqu'il atteint la condition prédéfinie de terminaison d'une boucle : il arrête de répéter la boucle. D'ailleurs, il n'est pas nécessaire de définir cette condition du tout ; on peut écrire une boucle qui se répète à l'infini par défaut, puis contrôler son comportement avec `break`.

```
var concatChaine = "a";  
while( true ) {  
    concatChaine = concatChaine + "a";  
    if( concatChaine.length === 5 ) {  
        break;  
    }  
    console.log( concatChaine );  
}  
aa  
aaa  
aaaa
```

`true` ne peut jamais cesser de renvoyer `true`, alors si on l'utilise comme condition d'une boucle `while`, celle-ci doit s'exécuter indéfiniment – à moins que nous n'utilisions `break` pour l'arrêter, une fois que notre chaîne atteint cinq caractères de long.

BOUCLES INFINIES

Nous touchons à un sujet épineux pour les développeurs de tout acabit : que se passe-t-il si nous utilisons une boucle `while` en prenant `true` comme condition, ou si nous écrivons une boucle `for` qui compte à partir de zéro, mais ne s'arrête que si `i` est égal à `-1` ? Nous venons de créer une boucle infinie, et d'ouvrir la boîte de Pandore.

Lorsqu'ils rencontrent une boucle infinie, certains navigateurs modernes finissent par vous offrir la possibilité d'interrompre le script – mais seulement les navigateurs récents, et pas toujours. Bien souvent, une boucle infinie fera planter le navigateur.

Ça arrive aux meilleurs d'entre nous, et il n'y a pas vraiment de mal (tant que le code fautif n'a pas eu le temps de se retrouver sur un site accessible au public). Fermez et rouvrez votre navigateur et vous pourrez reprendre vos activités habituelles ; veuillez simplement à corriger la boucle infinie avant d'essayer de relancer votre script.

ET MAINTENANT, TOUS ENSEMBLE

Nous avons abordé beaucoup de sujets jusqu'ici. Maintenant que nous avons une idée de la façon dont JavaScript gère les données, la logique et les boucles, nous pouvons commencer à assembler tous ces éléments pour produire quelque chose de plus utile que quelques lignes dans la console de développement. Il est temps d'écrire un script dans le contexte qu'il est censé occuper : une vraie page web.

5 SCRIPTER AVEC LE DOM

AVANT D'APPLIQUER NOS CONNAISSANCES sur une vraie page, revenons à un sujet entrevu au début de ce livre : le Document Object Model. Le DOM a deux fonctions : fournir à JavaScript une carte de tous les éléments de notre page et proposer un ensemble de méthodes pour accéder à ces éléments, à leurs attributs et à leur contenu.

La partie « objet » du Document Object Model devrait maintenant vous être plus familière que la première fois que nous avons abordé le DOM : le DOM est une représentation d'une page web sous la forme d'un objet, composée de propriétés qui représentent chaque élément enfant du document, des sous-propriétés qui représentent les enfants de chacun de ces enfants, et ainsi de suite. Une enfilade d'objets à perte de vue.

window : LE CONTEXTE GLOBAL

Tout ce que nous faisons avec JavaScript s'inscrit dans le cadre d'un seul objet : `window`. L'objet `window` représente, comme on pourrait s'y attendre, la fenêtre entière du navigateur. Il contient tout le DOM, mais aussi – et c'est là que ça se complique – l'intégralité de JavaScript.

Quand nous avons abordé la portée des variables, nous avons établi le concept de portées globale et locale, qui signifie qu'une variable peut être disponible dans tout notre script ou pour sa fonction parente uniquement.

C'est l'objet `window` lui-même qui représente cette portée globale. Toutes les fonctions et méthodes intégrées dans JavaScript sont construites à partir de l'objet `window`. Il n'est pas nécessaire de référencer `window` constamment, bien sûr, ou vous l'auriez déjà vu de nombreuses fois auparavant : comme `window` représente la portée globale, JavaScript vérifie `window` pour trouver toutes les variables que nous n'avons pas créées nous-mêmes. D'ailleurs, l'objet `console` que vous avez appris à aimer est une méthode de l'objet `window` :

```
window.console.log;  
function Log() { [native code] }
```

Il est difficile de visualiser ce que sont les variables à portée locale et globale avant de connaître `window`, mais beaucoup plus simple ensuite : quand on introduit une variable dans le cadre global, on en fait une propriété de `window` – et comme il n'est pas nécessaire de référencer explicitement `window` lorsqu'on souhaite accéder à l'une de ses propriétés ou de ses méthodes, on peut appeler cette variable n'importe où dans notre script en utilisant simplement son identifiant. Lorsqu'on accède à un identifiant, voici ce qui se passe en réalité :

```
function notreFonction() {  
    var localVar = "Je suis locale."  
    globalVar = "Je suis globale."  
    return "Je suis aussi globale !"  
};  
undefined  
window.notreFonction();  
Je suis aussi globale !  
window.localVar;
```

undefined

`window.globalVar;`

Je suis globale.

La représentation entière de la page par le DOM est une propriété de `window`, en l'occurrence `window.document`. Si vous saisissez simplement `window.document`; dans votre console, vous recevrez tout le balisage de la page actuelle sous la forme d'une seule chaîne énorme, ce qui n'est pas particulièrement utile – mais il est possible d'accéder à chaque élément de la page en tant que sous-propriété de `window.document`, exactement de la même façon. Souvenez-vous qu'il n'est pas nécessaire de spécifier `window` pour accéder à sa propriété `document` – `window` est unique en son genre.

`document.head;`

`<head>...</head>`

`document.body;`

`<body>...</body>`

Ces deux propriétés sont des objets qui contiennent eux-mêmes des propriétés qui sont des objets, et ainsi de suite le long de la chaîne. (Tout est un objet, ou presque.)

UTILISER LE DOM

Les objets de `window.document` constituent la carte du document en JavaScript, mais cette propriété n'est pas très utile pour nous – du moins, pas pour accéder aux nœuds du DOM comme nous accéderions à n'importe quel autre objet. Parcourir l'objet `document` manuellement serait un énorme casse-tête et nos scripts seraient immédiatement cassés à la moindre modification de notre balisage.

Mais `window.document` n'est pas simplement une représentation de la page ; c'est aussi une API plus intelligente pour accéder à ces informations. Par exemple, si nous voulons relever tous les éléments `p` d'une page, nous ne sommes pas obligés d'écrire toute une liste de propriétés. Nous utilisons une méthode intégrée de `document` qui les regroupe tous dans un tableau pour nous. Ouvrez le site de votre choix – du moment qu'il comporte vraisemblablement un élément paragraphe ou deux – et essayez de taper ça dans votre console :

```
document.getElementsByTagName( "p" );  
[<p>...</p>, <p>...</p>, <p>...</p>, <p>...</p>]
```

Comme nous travaillons avec des types de données qui nous sont familiers, nous savons déjà plus ou moins comment les utiliser :

```
var paragraphes = document.getElementsByTagName( "p" );  
undefined  
paragraphes.length;  
4  
paragraphes[ 0 ];  
<p>...</p>
```

Mais les méthodes du DOM ne produisent pas des tableaux, à proprement parler. Des méthodes comme `getElementsByTagName` renvoient des « listes de nœuds » qui se comportent essentiellement comme des tableaux. Chaque élément dans une `nodeList` correspond à un nœud individuel dans le DOM (un `p` ou un `div`, par exemple) et s'accompagne d'un certain nombre de méthodes intégrées spécifiques au DOM. Par exemple, la méthode `innerHTML` renverra tout le balisage qu'un nœud contient (éléments, texte, etc.) sous la forme d'une chaîne de caractères :

```
var paragraphes = document.getElementsByTagName( "p" ),  
    dernierIndice = paragraphes.length - 1, /* Utilise la  
    longueur de la liste de nœuds 'paragraphes' moins 1
```



```
pour obtenir le dernier paragraphe de la page */
dernierParagraphe = paragraphes[ dernierIndice ];
dernierParagraphe.innerHTML;
```

And that's how I spent my summer vacation.

De même que ces méthodes nous donnent accès aux informations de la page rendue, elles nous permettent de modifier ces informations. Par exemple, la méthode `innerHTML` permet de modifier des informations comme n'importe quel autre objet : un signe égal suivi de la nouvelle valeur.

```
var paragraphes = document.getElementsByTagName( "p" ),
    premierParagraphe = paragraphes[ 0 ];
premierParagraphe.innerHTML = "Listen up, chumps:";
"Listen up, chumps:"
```

La carte du DOM de JavaScript fonctionne dans les deux sens : la propriété `document` est actualisée chaque fois que le balisage change, et notre balisage est mis à jour à chaque modification de `document` (**FIG 5.1**).

De même, l'API DOM nous donne un certain nombre de méthodes permettant de créer, d'ajouter et de supprimer des éléments. Ces méthodes sont écrites dans un anglais plutôt facile à comprendre, quoiqu'un peu verbeux.

SCRIPTER AVEC LE DOM

Avant de commencer, abandonnons notre console de développement un instant. Il y a une éternité, nous avons créé un squelette de document HTML permettant de charger un script distant, et nous allons y revenir maintenant. Entre les connaissances que vous avez acquises sur JavaScript jusqu'à présent et notre introduction au DOM, nous n'en sommes plus à demander à notre console de tout répéter comme un vulgaire perroquet – le temps est venu de construire quelque chose de réel.

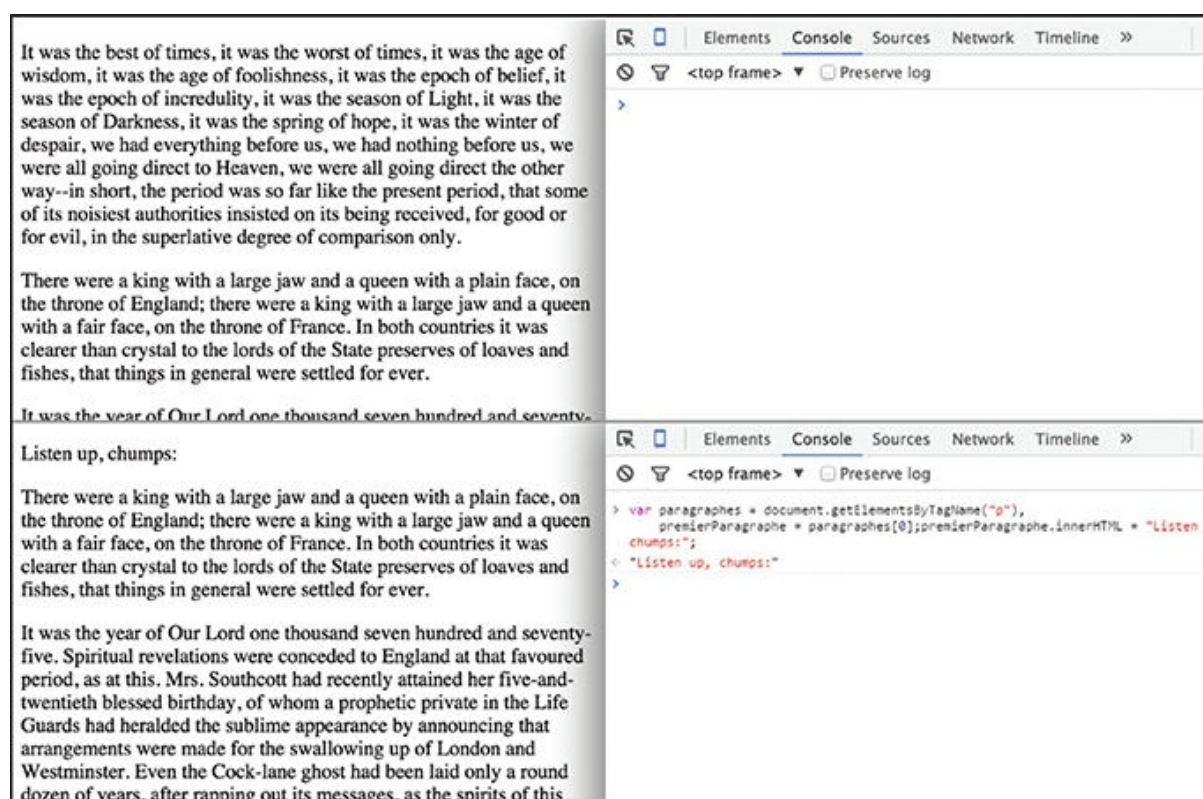


FIG 5.1 : Les premiers brouillons sont toujours difficiles...

Nous allons ajouter une « coupure » à une page d'index remplie de texte : un paragraphe d'accroche suivi d'un lien qui permet de révéler le texte complet. Mais nous n'allons pas obliger l'utilisateur à ouvrir une nouvelle page ; au lieu de ça, nous allons utiliser JavaScript pour afficher le texte complet sur la même page.

Commençons par créer un document HTML qui référence notre feuille de styles externe et notre fichier de script externe – rien de bien compliqué. Pour l'instant, la feuille de styles CSS et le fichier de script sont des fichiers vides portant l'extension .css et .js. J'aime garder mes feuilles de styles CSS dans un sous-répertoire /css et mes scripts dans un sous-répertoire /js, mais faites ce qui vous paraît le plus pratique.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css"
      href="css/style.css">
  </head>
  <body>
    <script src="js/script.js"></script>
  </body>
</html>
```

Nous allons remplir cette page avec plusieurs paragraphes de texte. N'importe quel bout de texte qui traîne dans le coin fera l'affaire, ou même (avec mes excuses aux experts en stratégie de contenu) un bon vieux lorem ipsum. Nous voulons simplement créer une esquisse d'article rapide, comme un billet de blog.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" type="text/css"
      href="css/style.css">
  </head>
  <body>
    <h1>JavaScript pour les web designers </h1>

    <p>EN TOUTE FRANCHISE, je devrais commencer ce livre par
des excuses – pas à vous, lecteur, quoique je vous en devrai
sans doute au moins une d'ici à la fin de ce livre. Non,
c'est à JavaScript que je dois des excuses, pour toutes les
ignominies que je lui ai dites au début de ma carrière – des
injures si corrosives qu'elles feraient fondre du verre
trempé.</p>

    <p>C'est ma façon peu subtile de dire que JavaScript peut
être un langage délicat à apprendre.</p>

    [ ... ]

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

N'hésitez pas à ouvrir la feuille de styles et à jouer un peu avec la typographie, mais ne vous éloignez pas trop du sujet. Il faudra écrire un peu de CSS par la suite, mais pour l'instant, nous avons un script à rédiger.

Nous pouvons découper ce script en plusieurs tâches individuelles : nous devons ajouter un lien Read more (Lire la suite) dans le premier paragraphe, cacher tous les éléments `p` à l'exception du premier et révéler ces éléments masqués lorsque l'utilisateur interagit avec le lien Read more.

Nous commencerons par ajouter le lien Read more à la fin du premier paragraphe. Ouvrez votre fichier script.js encore vide et saisissez le code suivant :

```
var nouveauLien = document.createElement( "a" );
```

Pour commencer, nous initialisons la variable `nouveauLien`, qui utilise la méthode `document.createElement ("a")` pour (vous l'aurez deviné) créer un nouvel élément `a`. Cet élément n'existe pas vraiment pour le moment ; nous devons l'ajouter manuellement pour le faire apparaître sur la page. Mais il ne sert pas à grand-chose d'ajouter des balises `<a>` sans attribut ni contenu. Avant de les ajouter sur la page, commençons par les remplir avec les informations dont elles ont besoin.

Nous pourrions faire cela après avoir ajouté le lien dans le DOM, bien sûr, mais pourquoi modifier plusieurs fois un élément de la page quand on peut obtenir le même résultat avec une seule modification ; en travaillant cet élément avant de le déposer sur la page, notre code reste prévisible.

Dans la mesure du possible, il est également préférable de n'accéder au DOM qu'une seule fois, en termes de performances – mais la micro-optimisation des performances peut rapidement devenir une obsession. Comme nous l'avons vu, JavaScript nous offre fréquemment plusieurs moyens de faire la même chose, et l'une de ces méthodes peut être techniquement plus performante que les autres. Ces considérations mèneront invariablement à un code « trop intelligent » – des boucles excessivement complexes qui demandent un exposé détaillé pour être décrites, tout ça pour raboter quelques picosecondes sur le temps de chargement. Je l'ai fait ; je me surprends encore parfois à le faire, mais essayez de vous en abstenir. Alors, s'il est une bonne habitude de limiter le nombre d'accès au DOM autant que possible pour des questions de performance, c'est aussi et surtout parce que cela permet de garder le code lisible et prévisible. En accédant au DOM uniquement quand on en a besoin, on évite de se répéter et on rend nos points d'interaction avec le DOM plus évidents pour les futurs programmeurs qui travailleront sur nos scripts.

Bon. Revenons à notre élément `<a>` vide et sans attributs qui flotte dans

le vide intersidéral de JavaScript, totalement détaché de notre document.

Nous pouvons maintenant utiliser deux autres interfaces du DOM qui rendront ce lien plus utile : `setAttribute` pour lui affecter des attributs, et `innerHTML` pour le remplir de texte. Ces deux instructions présentent une syntaxe légèrement différente. Avec `innerHTML`, on affecte simplement une chaîne de caractères, comme on attribuerait une valeur à n'importe quel autre objet. `setAttribute`, en revanche, requiert deux arguments : l'attribut et la valeur que nous souhaitons lui donner, dans cet ordre-là. Comme il ne faut pas que ce lien renvoie vers une page externe, nous nous contenterons d'un dièse comme `href` (un lien vers la page actuelle).

```
var nouveauLien = document.createElement( "a" );
nouveauLien.setAttribute( "href", "#" );
nouveauLien.innerHTML = "Lire la suite";
```

Vous remarquerez que nous utilisons ces interfaces sur notre référence à l'élément au lieu de l'objet `document` lui-même. Tous les nœuds du DOM ont accès à des méthodes comme celles que nous utilisons ici – nous utilisons uniquement `document.getElementsByTagName("p");` parce que nous voulons accéder à tous les paragraphes du document. S'il fallait seulement accéder aux paragraphes placés à l'intérieur d'un `div` particulier, nous pourrions faire la même chose avec une référence à ce `div` : `notreDiv.getElementsByTagName("p");`. Et comme nous cherchons à définir l'attribut `href` et l'HTML interne du lien que nous avons créé, nous référençons ces propriétés à l'aide de `nouveauLien.setAttribute` et de `nouveauLien.innerHTML`.

Ensuite, ce lien doit être placé à la fin de notre premier paragraphe, donc notre script aura besoin d'un moyen de référencer ce premier paragraphe. Nous savons déjà que `document.getElementsByTagName("p");` nous donne une liste de nœuds de tous les paragraphes de la page. Comme les listes de nœuds se comportent comme des tableaux, nous pouvons référencer le premier élément de la liste de nœuds en utilisant l'indice `0`.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( " p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
nouveauLien.setAttribute( "href", "#" );
nouveauLien.innerHTML = "Read more";
```

Pour la lisibilité du code, il est judicieux d'initialiser les variables au début du script, même sans leur attribuer immédiatement une valeur (`undefined`), si

nous avons l'intention de le faire par la suite. De cette façon, nous savons quels sont tous les identifiants utilisés.

Nous avons maintenant tout ce dont nous avons besoin pour ajouter un lien à la fin du premier paragraphe : l'élément à ajouter (`nouveauLien`) et l'élément auquel nous souhaitons l'ajouter (`premierParagraphe`).

L'une des méthodes intégrées à tous les nœuds du DOM est `appendChild`, qui permet d'ajouter un élément « enfant » à un nœud. Nous utiliserons cette méthode `appendChild` sur notre référence au premier paragraphe du document, en lui faisant passer `nouveauLien` comme argument.

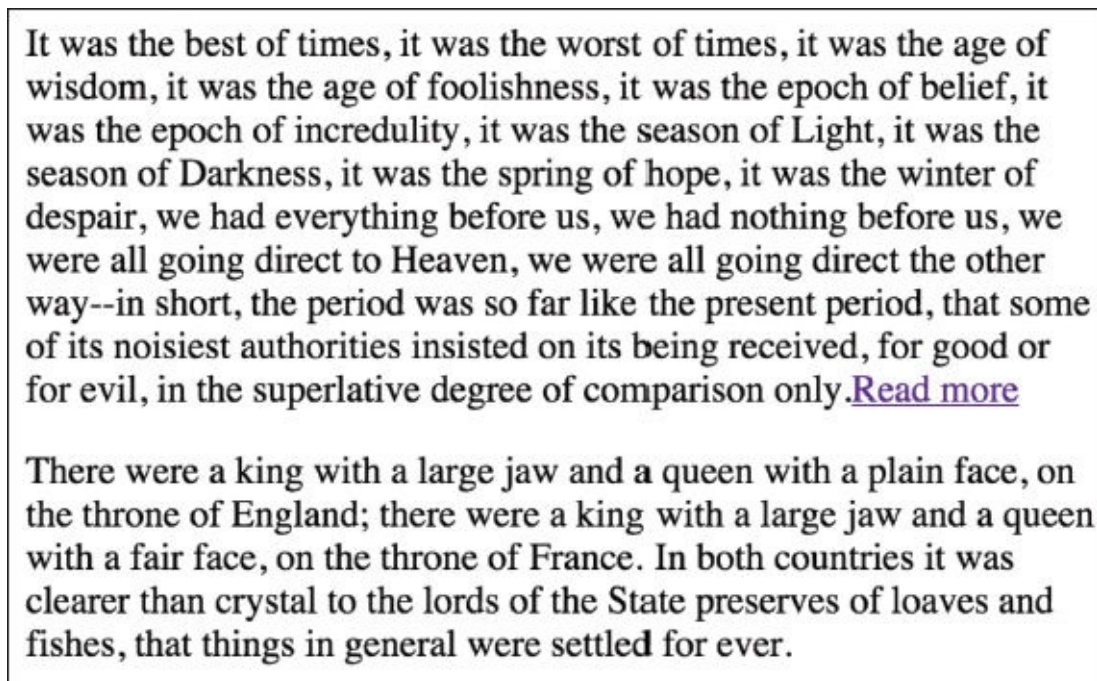


FIG 5.2 : Bon, c'est un début.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphe = document.getElementsByTagName
( " p " );
var premierParagraphe = tousLesParagraphe[ 0 ];
nouveauLien.setAttribute( "href", "#" );
nouveauLien.innerHTML = "Read more";
premierParagraphe.appendChild( nouveauLien );
```

Nous devrions enfin avoir quelque chose à cliquer une fois la page rechargée. Si tout se déroule convenablement, vous avez maintenant un lien Read more à la fin du premier paragraphe. Si rien ne se passe comme prévu – en raison d'un point-virgule mal placé ou de parenthèses décalées, par exemple, votre console de développement vous signalera qu'une erreur est survenue, alors n'oubliez pas de l'ouvrir.

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only. [Read more](#)

There were a king with a large jaw and a queen with a plain face, on the throne of England; there were a king with a large jaw and a queen with a fair face, on the throne of France. In both countries it was clearer than crystal to the lords of the State preserves of loaves and fishes, that things in general were settled for ever.

FIG 5.3 : Voilà, on s'en rapproche.

On y est presque, mais notre lien n'est pas idéalement placé : il se retrouve collé au paragraphe qui le précède, puisqu'il comporte l'attribut `display: inline` par défaut (FIG 5.2).

Nous avons plusieurs options pour régler ce problème : je n'aborderai pas ici toutes les différentes syntaxes, mais le DOM nous permet également d'accéder aux informations de style des éléments, quoiqu'à la base, il ne nous permette que de lire et de modifier des informations de style associées à un attribut `style`. Juste pour vous faire une idée de comment ça marche, changeons le lien en `display: inline-block` et ajoutons quelques pixels de marge à gauche, de sorte qu'il n'entre pas en collision avec notre texte. De même que nous avons défini les attributs, nous ferons tout ça avant d'ajouter le lien sur la page :

```
var nouveauLien = document.createElement( "a" );
var tousLesParagaphes = document.getElementsByTagName
( " p " );
var premierParagraphe = tousLesParagaphes[ 0 ];
nouveauLien.setAttribute( "href", "#" );
nouveauLien.innerHTML = "Read more";
nouveauLien.style.display = "inline-block";
nouveauLien.style.marginLeft = "10px";
premierParagraphe.appendChild( nouveauLien );
```

L'ajout de ces quelques lignes fonctionne, à une ou deux petites choses près.

Commençons par parler de la syntaxe (FIG 5.3).

Souvenez-vous que les identifiants ne peuvent pas comporter de tirets, et comme tout est un objet (ou presque), le DOM référence aussi les styles au format objet. Toute propriété CSS qui contient un tiret se retrouve ainsi convertie au format CamelCase : `margin-left` devient `marginLeft`, `border-radius-top-left` devient `borderRadiusTopLeft`, et ainsi de suite. Comme la valeur que nous affectons à ces propriétés est une chaîne de caractères, cependant, les tirets sont acceptables. Une bizarrerie de plus à retenir, mais cela reste raisonnable et ne devrait pas vous empêcher de styler des pages en JavaScript, si la situation l'exige.

Il y a une meilleure raison de ne pas modifier les styles avec JavaScript : préserver la séparation entre le comportement et la présentation. JavaScript est notre couche « comportementale », CSS est notre couche de présentation, et ces deux mondes doivent s'ignorer autant que possible. On doit pouvoir modifier des styles sur une page sans parcourir des lignes et des lignes de fonctions et de variables, tout comme on évite d'enfouir des styles dans notre code HTML. Les personnes qui se retrouveront à entretenir les styles de notre site ne seront pas forcément à l'aise avec JavaScript – et comme changer des styles en JavaScript consiste à ajouter indirectement des styles *via* des attributs `style`, tout ce que nous écrirons dans un script écrasera le contenu d'une feuille de styles par défaut.

Nous pouvons préserver cette séparation en utilisant plutôt `setAttribute` pour attribuer une classe à notre lien. Effaçons donc ces deux lignes de style et ajoutons-en une qui définit une classe à la place.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( " p " );
```


It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only. [Read more](#)

There were a king with a large jaw and a queen with a plain face, on the throne of England; there were a king with a large jaw and a queen with a fair face, on the throne of France. In both countries it was clearer than crystal to the lords of the State preserves of loaves and fishes, that things in general were settled for ever.

FIG 5.4 : Pas de changement visible, mais cette modification nous permet de garder nos décisions de style dans notre feuille de styles CSS et nos décisions comportementales en JavaScript

```
var premierParagraphe = tousLesParagraphe[ 0 ];
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
premierParagraphe.appendChild( nouveauLien );
```

Nous pouvons maintenant styler `.more-link` dans notre feuille de styles comme d'ordinaire :

```
.more-link {
    display: inline-block;
    margin-left: 10px;
}
```

C'est beaucoup mieux (**FIG 5.4**). Il est bon de se rappeler qu'en utilisant `setAttribute` de cette façon sur un nœud du DOM, on écrase toutes les classes déjà affectées à l'élément, mais ce n'est pas un problème si l'on crée cet élément à partir de zéro.

Nous sommes maintenant prêts à passer au deuxième élément de notre *to-do list* : masquer tous les autres paragraphes.

Comme nous avons modifié un code qui fonctionnait, n'oubliez pas de recharger la page pour vérifier que tout marche toujours comme prévu. Vous

ne devez pas introduire un bug à cette étape et continuer à écrire du code, sans quoi vous vous retrouverez inévitablement coincé à un moment et vous devrez revenir sur tous les changements que vous avez effectués. Si tout se passe comme prévu, la page ne devrait pas avoir changé visuellement lorsque vous la rechargez.

Nous avons maintenant une liste de tous les paragraphes de la page, et nous devons agir sur chacun d'entre eux. Il nous faut une boucle – et comme nous devons parcourir une liste de nœuds similaire à un tableau, nous utiliserons une boucle `for`. Juste pour nous assurer que notre boucle est en ordre, nous afficherons chaque paragraphe dans la console avant de poursuivre :

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
for( var i = 0; i < tousLesParagraphes.length; i++ ) {
    console.log( tousLesParagraphes[ i ] );
}
premierParagraphe.appendChild( nouveauLien );
```

▼ <p>
"It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only."
Read more
</p>

VM193:11

▼ <p>
"There were a king with a large jaw and a queen with a plain face, on the throne of England; there were a king with a large jaw and a queen with a fair face, on the throne of France. In both countries it was clearer than crystal to the lords of the State preserves of loaves and fishes, that things in general were settled for ever."
</p>

VM193:11

▼ <p>
"It was the year of Our Lord one thousand seven hundred and seventy-five. Spiritual revelations were conceded to England at that favoured period, as at this. Mrs. Southcott had recently attained her five-and-twentieth blessed birthday, of whom a prophetic private in the Life Guards had heralded the sublime appearance by announcing that arrangements were made for the swallowing up of

FIG 5.5 : Il semblerait que notre boucle fasse ce que l'on attend d'elle.

Votre lien Read more devrait toujours se trouver à la suite du premier paragraphe comme précédemment, et votre console devrait être pleine de texte (**FIG 5.5**).

Nous devons maintenant masquer ces paragraphes avec `display: none`, et nous avons pour cela plusieurs moyens à notre disposition : nous pourrions utiliser une classe comme nous l'avons fait avant, mais il ne serait pas totalement idiot d'appliquer des styles en JavaScript. Nous contrôlons l'action masquer/afficher depuis notre script, et il n'y a aucune chance pour qu'il faille remplacer ce comportement par autre chose dans une feuille de styles. Dans ce cas, il paraît judicieux d'utiliser les méthodes intégrées du DOM pour appliquer les styles :

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
for( var i = 0; i < tousLesParagraphes.length; i++ ) {
    tousLesParagraphes[ i ].style.display = "none";
}
premierParagraphe.appendChild( nouveauLien );
```

Si nous rechargeons maintenant la page, tout a disparu : notre JavaScript parcourt la liste entière des paragraphes et les masque tous. Nous devons ajouter une exception pour le premier paragraphe à l'aide d'une logique conditionnelle : une expression `if`, avec une variable `i` facile à évaluer :

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
for( var i = 0; i < tousLesParagraphes.length; i++ ) {
    if( i === 0 ) {
```

```
    continue;
}
    tousLesParagraphes[ i ].style.display = "none";
}
premierParagraphe.appendChild( nouveauLien );
```

Si c'est la première fois que la boucle s'exécute, l'instruction `continue` saute le reste de l'itération actuelle, puis la boucle passe à l'itération suivante (ce qui ne marcherait pas si on avait utilisé l'instruction `break`).

Si vous rechargez maintenant la page, vous verrez un seul paragraphe avec un lien Read more, et tous les autres paragraphes seront masqués. Tout a l'air de bien se passer jusqu'ici, et si ça ne se passe pas si bien que ça pour vous, vérifiez votre console pour vous assurer qu'il n'y a rien à corriger.

ÉVÉNEMENTS DOM

Bon, il nous reste une dernière chose à faire : nous voulons que notre lien Read more fasse quelque chose. Si vous cliquez dessus maintenant, il vous ramènera en haut de la page actuelle et ajoutera un dièse à l'URL.

Pour ce qui est d'être à l'épreuve des bugs, ce script offre de bonnes garanties. Quel que soit l'incident qui puisse se produire par la suite (erreur dans l'un de nos scripts, dans un script tiers qui échappe à notre contrôle, ou même dans le navigateur de l'utilisateur), le texte complet sera visible.

Nous insérons le lien Read more avec JavaScript au lieu de le coder en dur dans le balisage, de sorte que si JavaScript n'est pas disponible pour une raison ou pour une autre, l'utilisateur ne verra pas un lien inutile flottant à la fin du premier paragraphe. Nous utilisons également JavaScript pour ajouter la classe qui masque les autres paragraphes plutôt que d'ajouter une classe dans le balisage et de les masquer à l'aide des feuilles de styles ; ainsi, même si un script faisait défaut, le contenu serait toujours disponible pour l'utilisateur.

L'idée de commencer avec quelque chose d'utilisable et d'ajouter des couches d'amélioration en JavaScript sur cette base de référence s'appelle amélioration progressive, et nous y reviendrons sous peu. Pour le moment, nous avons un script à finir.

Dans les faits, les événements DOM sont une API régissant l'activité qui se déroule dans le navigateur. Cela inclut les actions de l'utilisateur, les animations CSS et les événements internes du navigateur, par exemple le moment où une image a fini de se charger, pour n'en nommer qu'un.

Ici, nous voulons simplement rédiger un comportement pour les utilisateurs qui cliquent sur notre lien généré. Nous n'avons pas besoin d'accéder une deuxième fois au DOM ni de parcourir chaque lien de la page – nous avons déjà une référence à notre lien, et nous allons utiliser une méthode intégrée du DOM pour détecter les événements : `addEventListener`.

Commençons par écrire notre fonction : que voulons-nous qu'il se passe lorsqu'on clique sur le lien ?

Eh bien, tout d'abord, nous voulons afficher tous les paragraphes masqués sur la page, donc nous devons rétablir les styles `display: block`. Une fois que nous aurons affiché tous ces paragraphes, le lien Read more aura perdu son utilité pour l'utilisateur – il faudra donc supprimer ce lien du DOM.

Nous allons créer une nouvelle fonction portant l'identifiant `afficheTexte`, et

pour l'instant, nous placerons une instruction `console.log` dans cette fonction pour nous assurer que tout fonctionne correctement. Nous utiliserons ensuite `addEventListener` sur `nouveauLien` pour détecter les événements de clic. `addEventListener` prend deux arguments : une chaîne définissant le type d'événement que nous voulons détecter (dans ce cas, `"click"`) et la fonction à exécuter lorsque cet événement se produit.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
function afficheTexte() {
    console.log( "Cliqué !" );
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", afficheTexte );
for( var i = 0; i < tousLesParagraphes.length; i++ ) {
    if( i === 0 ) {
        continue;
    }
    tousLesParagraphes[ i ].style.display = "none";
}
premierParagraphe.appendChild( nouveauLien );
```

Jusque-là, tout va bien ! Cliquez sur notre lien généré et la console renverra la chaîne `Cliqué !`.

Le navigateur suit toujours le lien, cependant. Ce n'est pas vraiment un problème pour le moment, puisque le `href` de notre lien est un dièse, mais nous voudrions parfois ajouter un comportement personnalisé sur un vrai lien, sans que le navigateur renvoie l'utilisateur vers une nouvelle page au lieu d'afficher notre comportement.

Par chance, `addEventListener` nous donne des informations sur l'événement de clic d'un utilisateur, sous la forme (vous l'aurez deviné) d'un objet. Et comme on pourrait s'y attendre, l'objet contient un certain nombre de propriétés à propos de l'événement, ainsi que de méthodes qui servent à contrôler le comportement du navigateur. Cet objet d'événement s'utilise sous la forme d'un argument, mais nous ne pouvons pas l'utiliser tant que nous ne

lui avons pas attribué un identifiant – la convention courante est de le nommer `e`, pour « événement ». Ajoutons cela comme argument, et modifions notre `console.log` pour afficher cet objet :

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
function afficheTexte( e ) {
    console.log( e );
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", afficheTexte );
for( var i = 0; i < tousLesParagraphes.length; i++ ) {
    if( i === 0 ) {
        continue;
    }
    tousLesParagraphes[ i ].style.display = "none";
}
premierParagraphe.appendChild( nouveauLien );
```

Lorsque nous cliquons sur ce lien généré, notre console renvoie un objet mystérieux :

```
MouseEvent {dataTransfer: null, which: 1, toElement: a.more-
link, fromElement: null, y: 467...}
```

Il se passe pas mal de choses là-dedans, mais une seule méthode de cet objet nous intéresse : `e.preventDefault()`, qui bloque le comportement par défaut du navigateur lorsqu'un événement se produit (dans ce cas, suivre le lien). Cette fonction peut apparaître n'importe où dans la fonction qui est liée à un événement : du moment qu'elle est présente quelque part dans `afficheTexte`, le navigateur n'essaiera pas de suivre notre lien.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
function afficheTexte( e ) {
```

```

    e.preventDefault();
};
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", afficheTexte );
for( var i = 0; i < tousLesParagraphe.length; i++ ) {
    if( i === 0 ) {
        continue;
    }
    tousLesParagraphe[ i ].style.display = "none";
}
premierParagraphe.appendChild( nouveauLien );

```

Désormais, le dièse dans l'attribut `href` du lien sera complètement ignoré. Même si nous changions `nouveauLien.setAttribute("href", "#");` pour renvoyer vers une vraie URL, vous pourriez cliquer sur le lien et le navigateur ne vous amènerait pas sur la page. Parfait ! Maintenant, il faut juste que notre fonction... fasse quelque chose.

Comme nous devons remplacer la propriété `display` de chaque paragraphe masqué par une valeur visible (comme `block`), il faut les parcourir à nouveau à l'aide d'une boucle. Pour l'instant, contentons-nous de copier et coller notre variable `tousLesParagraphe` et la boucle correspondante dans la fonction, et de remplacer la valeur de `display` par `"block"`.

```

var nouveauLien = document.createElement( "a" );
var tousLesParagraphe = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphe[ 0 ];
function afficheTexte( e ) {
    var tousLesParagraphe = document.getElementsByTagName
    ( "p" );
    for( var i = 0; i < tousLesParagraphe.length; i++ ) {
        if( i === 0 ) {
            continue;
        }
        tousLesParagraphe[ i ].style.display = "block";
    }
}

```



```

        e.preventDefault();
    }
    nouveauLien.setAttribute( "href", "#" );
    nouveauLien.setAttribute( "class", "more-link" );
    nouveauLien.innerHTML = "Read more";
    nouveauLien.addEventListener( "click", afficheTexte );
    for( var i = 0; i < tousLesParagraphe.length; i++ ) {
        if( i === 0 ) {
            continue;
        }
        tousLesParagraphe[ i ].style.display = "none";
    }
    premierParagraphe.appendChild( nouveauLien );

```

Copier-coller le code de cette façon n'est pas très DRY, mais ça marche, et nous pourrions nettoyer tout ça en temps utile. Testez le script et vous verrez que vous en avez presque fini : tous les paragraphes sauf le premier sont masqués une fois la page chargée, et ils sont révélés lorsque nous cliquons sur ce lien.

Il nous reste une petite chose à faire : il faut retirer le lien Read more du DOM une fois qu'il a été cliqué, puisqu'il ne sert plus à rien. L'opération est facile : une méthode `remove` est intégrée dans chaque nœud du DOM.

Pour commencer, cependant, nous avons besoin d'une référence au lien à supprimer. Inutile d'accéder de nouveau au DOM pour cela : l'instruction `this` dans une fonction attachée à un événement désigne l'élément qui a initié l'événement. Dans `afficheTexte`, `this` désigne notre nœud Read more, et nous allons lui appliquer la méthode `remove()` :

```

var nouveauLien = document.createElement( "a" );
var tousLesParagraphe = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphe[ 0 ];
function afficheTexte( e ) {
    var tousLesParagraphe = document.getElementsByTagName
    ( "p" );
    for( var i = 0; i < tousLesParagraphe.length; i++ ) {
        if( i === 0 ) {
            continue;

```

```

    }
    tousLesParagraphe[ i ].style.display = "block";
}
this.remove();
e.preventDefault();
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", afficheTexte );
for( var i = 0; i < tousLesParagraphe.length; i++ ) {
    if( i === 0 ) {
        continue;
    }
    tousLesParagraphe[ i ].style.display = "none";
}
premierParagraphe.appendChild( nouveauLien );

```

Et ça marche ! Nous avons un produit minimum viable ; ce n'est pas le code le plus élégant, mais nous avons construit exactement ce que nous voulions construire. Ça ne mérite peut-être pas une Légion d'honneur, mais au moins une épingle à la boutonnière.

Maintenant, nous pouvons optimiser.

Souvenez-vous de la boucle que nous avons copiée-collée : nous avons là une certaine marge de progression. Nous avons déjà une fonction qui parcourt tous nos paragraphes et change la propriété `display` de tous ces paragraphes, sauf le premier, et les fonctions sont là pour être réutilisées. Comme cette propriété `display` doit changer dans deux situations (pour prendre la valeur `none` initialement, puis `block` lorsque notre lien sera cliqué), nous allons retravailler cette fonction pour qu'elle serve dans ces deux cas de figure.

Pour commencer, nous devons changer le nom de cette fonction. Nous ne l'utiliserons pas seulement pour révéler nos paragraphes, mais aussi pour les masquer. Comme nous modifions la visibilité de ces paragraphes, nous allons changer l'identifiant et mettre à jour la référence correspondante à l'intérieur d'`addEventListener` ; j'ai choisi de l'appeler `basculeTexte`. Essayons ensuite d'appeler cette fonction à la place de notre boucle d'origine :

```

var nouveauLien = document.createElement( "a" );

```

```

var tousLesParagraphe = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphe[ 0 ];
function basculeTexte( e ) {
    var tousLesParagraphe = document.getElementsByTagName
    ( "p" );
    for( var i = 0; i < tousLesParagraphe.length; i++ ) {
        if( i === 0 ) {
            continue;
        }
        tousLesParagraphe[ i ].style.display = "block";
    }
    this.remove();
    e.preventDefault();
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", basculeTexte );
basculeTexte();
premierParagraphe.appendChild( nouveauLien );

```

Oups.

Uncaught TypeError: this.remove is not a function

Nous avons incorporé plusieurs hypothèses risquées dans notre fonction. JavaScript s'attend à ce que `this` référence un nœud du DOM qui comporte une méthode `remove()` ; or, cela ne s'appliquera pas à l'extérieur de notre événement. Notre script n'est même pas parvenu à la ligne d'après, qui aurait elle aussi engendré une erreur : là encore, la fonction prévoit un argument `e` avec une méthode `preventDefault` attachée, qui n'existe pas si nous n'appelons pas cette fonction en réponse à un événement. `e` porte un identifiant, mais sans `addEventListener` pour lui attribuer un objet, il n'est qu'un identifiant contenant `undefined`.

Nous allons commencer par nous occuper de l'erreur dans notre console. Nous devons vérifier que `this` est une référence au lien que nous avons créé, et le cas échéant, le supprimer. Rien de plus simple : nous avons déjà une référence au lien `Read more`, l'identifiant `nouveauLien`. Nous allons

simplement nous assurer que `this` et `nouveauLien` sont égaux.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphe = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphe[ 0 ];
function basculeTexte( e ) {
    var tousLesParagraphe = document.getElementsByTagName(
        "p" );
    for( var i = 0; i < tousLesParagraphe.length; i++ ) {
        if( i === 0 ) {
            continue;
        }
        tousLesParagraphe[ i ].style.display = "block";
    }
    if( this === nouveauLien ) {
        this.remove();
    }
    e.preventDefault();
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", basculeTexte );
basculeTexte();
premierParagraphe.appendChild( nouveauLien );
```

Voilà une erreur de moins, mais nous nous heurtons maintenant au problème de `preventDefault`, comme prévu :

```
Uncaught TypeError: Cannot read property
'preventDefault' of undefined
```

Lorsque nous invoquons `basculeTexte` sans argument, `e` prend la valeur `undefined`, et `undefined` ne comporte évidemment pas de méthode `preventDefault`. Cette valeur `undefined` par défaut signifie que l'identifiant `e` nous donne juste ce dont nous avons besoin pour que notre fonction soit remise en état de fonctionnement : une condition à tester. Nous invoquerons uniquement `e.preventDefault` si `e` ne comporte pas la valeur `undefined` :

```

var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
function basculeTexte( e ) {
    var tousLesParagraphes = document.getElementsByTagName
    ( "p" );
    for( var i = 0; i < tousLesParagraphes.length; i++ ) {
        if( i === 0 ) {
            continue;
        }
        tousLesParagraphes[ i ].style.display = "block";
    }
    if( this === nouveauLien ) {
        this.remove();
    }
    if( e !== undefined ) {
        e.preventDefault();
    }
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", basculeTexte );
basculeTexte();
premierParagraphe.appendChild( nouveauLien );

```

Les erreurs sont corrigées, mais nous nous avançons encore beaucoup : si nous passons un argument à cette fonction en l'invoquant en dehors d'un événement, `e` n'aurait pas la valeur `undefined`. `e` prendrait la valeur de l'argument, qui n'aurait probablement pas de méthode `preventDefault`, et nous nous retrouverions avec une erreur de plus. Nous ne sommes pas idiots au point de balancer un argument errant dans `basculeTexte`, car enfin, nous avons construit ce script – nous savons que cela n'apporterait rien de bon. Mais il n'y a certainement pas de mal à corriger quelques erreurs potentielles pour quiconque devra entretenir notre code après nous.

Pour plus de sécurité, nous allons rendre notre expression conditionnelle un

peu plus explicite : tout d'abord, nous vérifierons s'il existe un argument. Le cas échéant, nous verrons si cet argument comporte une méthode `preventDefault`. Comme nous évaluons deux valeurs qui doivent toutes deux renvoyer `true`, nous utiliserons l'opérateur `&&`.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagrapes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagrapes[ 0 ];
function basculeTexte( e ) {
    var tousLesParagrapes = document.getElementsByTagName
    ( "p" );
    for( var i = 0; i < tousLesParagrapes.length; i++ ) {
        if( i === 0 ) {
            continue;
        }
        tousLesParagrapes[ i ].style.display = "block";
    }
    if( this === nouveauLien ) {
        this.remove();
    }
    if( e !== undefined && e.preventDefault !== undefined ) {
        e.preventDefault();
    }
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", basculeTexte );
basculeTexte();
premierParagraphe.appendChild( nouveauLien );
```

Toujours pas d'erreur, tout semble bien se passer.

Cependant, aucun paragraphe n'est masqué – la propriété `display` prend toujours la valeur `block`. Ce que nous devons faire, c'est affecter la valeur `block` à ces éléments uniquement s'ils sont déjà masqués – nous aurons besoin d'un `if` de plus pour vérifier si la propriété `display` du paragraphe est

`none`, et dans le cas contraire, lui attribuer la valeur `block`. Pour toute autre valeur, nous lui affecterons la valeur `none`.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphes = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphes[ 0 ];
function basculeTexte( e ) {
    var tousLesParagraphes = document.getElementsByTagName
    ( "p" );
    for( var i = 0; i < tousLesParagraphes.length; i++ ) {
        if( i === 0 ) {
            continue;
        }
        if( tousLesParagraphes[ i ].style.display === "none"
        ) {
            tousLesParagraphes[ i ].style.display = "block";
        } else {
            tousLesParagraphes[ i ].style.display = "none";
        }
    }
    if( this === nouveauLien ) {
        this.remove();
    }
    if( e !== undefined && e.preventDefault !== undefined ) {
        e.preventDefault();
    }
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", basculeTexte );
basculeTexte();
premierParagraphe.appendChild( nouveauLien );
```

Ça marche à nouveau ! Une petite chose de plus, toutefois : nous ne cessons de répéter `tousLesParagraphes[i]` alors que nous pourrions le référencer à

l'aide d'un simple identifiant.

Je pinaille un peu, mais ça ne peut pas faire de mal.

```
var nouveauLien = document.createElement( "a" );
var tousLesParagraphe = document.getElementsByTagName
( "p" );
var premierParagraphe = tousLesParagraphe[ 0 ];
function basculeTexte( e ) {
    var tousLesParagraphe = document.getElementsByTagName
    ( "p" );
    for( var i = 0; i < tousLesParagraphe.length; i++ ) {
        var para = tousLesParagraphe[ i ];
        if( i === 0 ) {
            continue;
        }
        if( para.style.display === "none" ) {
            para.style.display = "block";
        } else {
            para.style.display = "none";
        }
    }
    if( this === nouveauLien ) {
        this.remove();
    }
    if( e !== undefined && e.preventDefault !== undefined ) {
        e.preventDefault();
    }
}
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", basculeTexte );
basculeTexte();
premierParagraphe.appendChild( nouveauLien );
```

Et tant qu'on y est, nous polluons légèrement la portée globale – toutes ces

variables sont ajoutées à `window`, comme elles ne sont pas englobées dans une fonction. Essayez vous-même dans la console :

```
window.nouveauLien;  
<a></a>
```

Pour éviter d'encombrer la portée globale, nous allons emballer le tout dans une fonction ; et comme il ne faut pas la référencer en dehors du moment où la page est chargée, nous n'aurons pas besoin de lui donner un identifiant – nous allons emballer tout notre JavaScript dans une fonction anonyme qui s'exécute immédiatement, ce que l'on appelle une expression fonctionnelle immédiatement invoquée, ou IIFE (*immediately-invoked functional expression*). La syntaxe est un peu étrange et peut être écrite de différentes manières, mais pour l'essentiel, voilà ce qui se passe : en emballant une fonction anonyme entre parenthèses, nous indiquons à JavaScript que toute instance de l'instruction `function` est une expression, et non pas une déclaration – autrement dit, elle invoque une fonction, sans nécessairement la définir avec un identifiant. Nous faisons suivre la fonction d'une paire de parenthèses – `()` – pour exécuter immédiatement cette fonction fraîchement créée.

Honnêtement, nous abordons un terrain légèrement académique : le modèle IIFE est important, c'est certain, mais nous n'avons pas besoin de savoir comment JavaScript traite les parenthèses pour nous en servir. Pour l'instant, nous pouvons le prendre pour argent comptant.

```
(function() {  
    var nouveauLien = document.createElement( "a" );  
    var tousLesParagraphe = document.getElementsByTagName  
        ( "p" );  
    var premierParagraphe = tousLesParagraphe[ 0 ];  
    function basculeTexte( e ) {  
        var tousLesParagraphe = document.  
            getElementsByTagName( "p" );  
        for( var i = 0; i < tousLesParagraphe.length; i++ ) {  
            var para = tousLesParagraphe[ i ];  
            if( i === 0 ) {  
                continue;  
            }  
            if( para.style.display === "none" ) {  
                para.style.display = "block";  
            }  
        }  
    }  
})
```

```

    } else {
        para.style.display = "none";
    }
}
if( this === nouveauLien ) {
    this.remove();
}
if( e !== undefined && e.preventDefault !== undefined ) {
    e.preventDefault();
}
};
nouveauLien.setAttribute( "href", "#" );
nouveauLien.setAttribute( "class", "more-link" );
nouveauLien.innerHTML = "Read more";
nouveauLien.addEventListener( "click", basculeTexte );
basculeTexte();
premierParagraphe.appendChild( nouveauLien );
}());

```

Maintenant, si nous tapons `window.nouveauLien;` dans notre console de développement, celle-ci renvoie *undefined* – nous ne polluons pas la portée globale avec des identifiants auxquels nous n’aurons jamais besoin d’accéder en dehors de notre IIFE.

```

window.nouveauLien;
undefined

```

Parfait.

Enfin non, pas parfait. Parfait, ça n’existe pas. Il y a toujours moyen de figoler un script avec de petites optimisations successives, *ad infinitum*. Mais c’est déjà pas mal, si j’ose dire : la portée globale n’est pas polluée, le code est raisonnablement DRY, et ce que nous avons écrit sera facile à lire et à entretenir longtemps après que nous serons passés à des scripts plus gros et plus beaux.

AMÉLIORATION PROGRESSIVE

Il n'est pas toujours simple de scripter des comportements de façon responsable. Nous détournons le comportement du navigateur et prenons le contrôle de l'expérience de l'utilisateur pour quelque chose d'aussi courant et prévisible que de cliquer sur un lien. Si c'est fait de manière discrète, nous créons une expérience complètement fluide – meilleure, bien souvent, que celle que le navigateur est capable d'offrir.

Mais sans une approche responsable, nous faisons bien pire que de simplement présenter un `div` mal aligné – nous construisons quelque chose qui pourrait s'avérer totalement inutilisable. Le Web est un support imprévisible, et nous devons prendre en compte cet état de fait ; encore plus, bien plus avec JavaScript qu'HTML ou CSS.

Nous aurions pu prendre quelques raccourcis dans le script d'affichage/masquage de paragraphe que nous avons rédigé aujourd'hui, par exemple. Nous aurions pu masquer ces paragraphes dès le départ en utilisant CSS et nous reposer sur JavaScript pour les afficher de nouveau, ou bien coder le lien Read more en dur et espérer que la fonctionnalité de notre script soit toujours disponible. Ce dernier cas serait une vraie nuisance si le moindre problème venait à se produire – une erreur ailleurs dans un script faisant planter le nôtre, par exemple. L'utilisateur se retrouverait alors avec un lien Read more qui ne fait rien. Le premier cas serait encore plus désastreux : à la moindre erreur de JavaScript, l'utilisateur se retrouverait sans aucun moyen d'accéder au contenu de la page.

Un site qui se repose entièrement sur JavaScript pour ses fonctionnalités critiques, qui est bâti sur le postulat que JavaScript sera toujours disponible, est un site fragile. Les conditions de navigation des utilisateurs peuvent changer d'une minute à l'autre, et nous ne pouvons pas prévoir – nous ne pouvons pas savoir – quels problèmes potentiels sont susceptibles de casser nos scripts.

Il y a quelques années, j'ai travaillé sur le site responsive du *Boston Globe* avec Ethan Marcotte, Scott Jehl et toute l'équipe du Filament Group. Nous l'avons entièrement construit sur la base de l'amélioration progressive, ce qui ne nous a pas restreints, bien au contraire ; ce site comporte quelques fonctionnalités incroyables, sans exagérer (<http://www.bostonglobe.com/>).

Nous avons dû résoudre quelques problèmes épineux, mais nous nous sommes attachés à appliquer le concept d'amélioration progressive à la lettre : « Si cette fonctionnalité n'est pas opérationnelle, comment faire en sorte que

l'utilisateur ait tout de même accès aux informations sous-jacentes ? » En surface, cela peut sembler être un exercice en cas extrêmes. Les décisions infimes qui sont prises au cours de la création d'un site web ne semblent pas forcément très importantes sur le coup.

Les attentats du marathon de Boston se sont produits quelques années plus tard. Ce jour-là, il était extrêmement important pour un grand nombre de personnes de pouvoir accéder à des informations en temps réel sur ce qu'il se passait à travers la ville, et beaucoup d'entre eux se sont tournés vers le site du *Boston Globe*. En raison du pic de trafic, le CDN du *Boston Globe* – le serveur qui transmettait les ressources du site, comme les feuilles de styles et les scripts – s'est retrouvé surchargé, et est tombé en panne. Pendant un moment cet après-midi, le site du *Boston Globe* était disponible en HTML uniquement.

Le site web était manifestement déficient, et aucune de nos fonctionnalités JavaScript avancées ne marchait correctement : pas de lecture hors ligne, pas de menus déroulants dans la barre de navigation. Parfois, le site tout entier se limitait à du texte noir en Times New Roman sur fond blanc. Parfois, seules les feuilles de styles étaient rétablies, et pas toujours dans leur intégralité – idem avec le code JavaScript. Nos scripts s'exécutaient rarement sans erreur, bien que nous n'ayons commis aucune faute de notre côté.

Mais cet après-midi-là, les visiteurs de BostonGlobe.com ont tout de même pu visiter le site. Ils ont pu lire les nouvelles. Le site fonctionnait. Si nous nous étions reposés sur CSS pour masquer des parties de la navigation et avions compté sur le fait que JavaScript serait toujours présent pour les afficher de nouveau, certains utilisateurs n'auraient pas pu consulter le site ce jour-là. Si nous nous étions fiés à JavaScript pour charger et afficher des morceaux critiques de la page, ce contenu ne serait peut-être jamais apparu. Si nous avions codé des commandes en HTML qui nécessitaient JavaScript pour fonctionner, celles-ci auraient été inutiles – frustrant pour les utilisateurs du site, au pire moment possible.

L'amélioration progressive est une chose de plus à prendre en compte lorsque nous écrivons nos scripts. Je ne vous mentirai pas, l'amélioration progressive vous demandera souvent un peu plus de travail – mais c'est le métier qui veut ça. Certaines décisions qui ont été prises en codant les scripts de BostonGlobe.com auraient pu sembler anodines, tout bien considéré ; mais ce jour-là, pour des dizaines de milliers d'utilisateurs, ces décisions cumulées ont pris toute leur importance. Pour ces utilisateurs, l'amélioration progressive a fait la différence entre trouver l'information dont ils avaient besoin à ce moment précis, ou être obligé de continuer à chercher. Savoir ou ne pas savoir.

CONCLUSION

Nous avons fait du chemin, depuis « Hello, world » jusqu'à coder un script qui utilise les interactions de l'utilisateur pour modifier une page entière.

Cela fait un sacré paquet de JavaScript en bien peu de pages, alors si vous n'avez pas retenu chaque définition ou tapé chaque petit bout de code dans votre console, ne vous en faites pas. Le but de ce livre n'était pas de parvenir à la maîtrise totale de JavaScript, après tout. Comme promis, nous n'avons fait qu'en effleurer les possibilités. Et JavaScript, comme toute autre norme du Web, est en évolution constante – nous ne pouvons pas tout savoir, et nous n'en aurons jamais besoin. Encore aujourd'hui, je passe beaucoup de temps sur le Web pour trouver le meilleur moyen de faire telle ou telle chose avec JavaScript.

Mais même avec toutes ces nouvelles API, ces fonctions et ces pouvoirs que JavaScript nous confère (ou nous confèrera un jour), les bases ne changeront pas. Un tableau sera toujours un tableau ; un événement sera un événement. Tout ce que vous avez lu dans ce livre, vous saurez l'appliquer et le reconnaître lorsque vous lirez un script codé par quelqu'un d'autre.

Vous n'aurez pas soudainement un cerveau de programmeur-cyborg après avoir fini ce livre, et de toute façon, ce n'est pas ça qui fait un développeur. Ce qui fait un développeur, c'est la curiosité, l'envie d'apprendre, et peut-être de résoudre quelques énigmes.

Si vous êtes arrivé jusqu'ici, vous êtes déjà en bonne voie.

RESSOURCES

Alors, où aller, maintenant ? Un langage de programmation tout entier s'offre à vous. Il y a de nombreux autres livres à lire (et vous êtes invité à les lire), mais maintenant que vous savez à quoi vous attendre, il n'y a rien de tel que de lire du code. Prenez un peu de temps pour parcourir les scripts que vous avez vu passer sur un projet au travail, ou explorez le code qui fait fonctionner vos outils open source favoris. Vous y trouverez beaucoup plus de choses que ce que nous avons abordé ici, mais je parie que vous en reconnaîtrez plus que ce que vous pensez.

Étapes suivantes

- **Mozilla Developer Network.** Si vous êtes arrivé ici, il est temps que je vous révèle le secret d'une carrière fructueuse dans le développement JavaScript professionnel : la triche. JavaScript est trop complexe pour être retenu par un simple mortel. Quand vous vous posez une question, qu'il s'agisse de la capitalisation ou de la compatibilité des navigateurs, MDN est *la* ressource à exploiter (<http://bkaprt.com/jsfwd/07-01/>).
- **Design web responsive et responsable**, Scott Jehl. L'amélioration progressive est un concept que nous avons à peine effleuré, mais il forme la base du script que nous avons construit. Les améliorations en JavaScript ne sont pas toujours accessibles à vos utilisateurs, et pas seulement parce que certains le désactivent dans leur navigateur. L'ouvrage *Design web responsive et responsable* de la collection A Book Apart (paru en français aux éditions Eyrolles) est une étude approfondie de pratiques de développement front-end inclusives, de l'amélioration progressive à l'accessibilité en passant par l'optimisation des performances (<http://bkaprt.com/jsfwd/07-02/>).
- **If Hemingway Wrote JavaScript**, Angus Croll. Vous ne l'aurez peut-être pas remarqué si vous suivez mon compte Twitter, mais je suis un grand fan de littérature. *If Hemingway Wrote Javascript* (« Si Hemingway écrivait en JavaScript ») est un livre qui tente d'imaginer les styles de programmation d'auteurs tels que Jane Austen et William Shakespeare sur la base de leurs écrits. Le livre ne contient pas que du JavaScript, mais tous les langages que vous y trouverez se composent des principes que vous avez étudiés dans ce livre. C'est un moyen amusant de voir les signatures que les gens laissent dans leur code (<http://bkaprt.com/jsfwd/07-03/>).

Pour approfondir

Vous connaissez maintenant suffisamment les bases pour vous mettre à JavaScript à fond. Si vous frétillez à l'idée d'approfondir vos connaissances et de démêler des énigmes en JavaScript, j'ai d'autres ressources écrites à partager avec vous (en anglais).

- **Eloquent JavaScript**, Marijn Haverbeke. La substance de ce livre ne devrait pas vous paraître complètement étrangère, à moins que vous n'ayez pas vraiment lu celui que vous tenez entre vos mains : Marijn trace un parcours similaire à travers les concepts de JavaScript, mais de façon un peu plus rapide et détaillée (<http://eloquentjavascript.net/>).
- **JavaScript Patterns**, Stoyan Stefanov. Je suis revenu à ce livre plusieurs fois au cours de ma carrière. Je le parcourais jusqu'à avoir l'impression d'être un peu perdu, puis je le reposais. Avec le temps et l'expérience, j'avais un peu plus loin à chaque fois, et ce faisant, je trouvais d'innombrables moyens de réexaminer des parties de JavaScript que je croyais connaître sur le bout des doigts (<http://bkaprt.com/jsfwd/07-04/>).
- **Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript**, David Herman. Comme d'autres livres de cette liste, *Effective JavaScript* n'est pas une lecture très légère. Il constitue toutefois une suite extrêmement abordable des concepts que nous avons abordés ensemble, le tout compilé sous la forme d'une liste indispensable de bonnes pratiques et de conseils utiles (<http://effectivejs.com/>).

REMERCIEMENTS

Mince, je n'aurais jamais cru écrire autant. Il en va de même pour le reste de ce livre, franchement. Mais j'imagine que j'aurais dû le savoir. J'aurais dû voir tout ça venir.

Pas grâce à moi, grands dieux non, mais grâce aux gens qui m'ont aidé à écrire, à m'exprimer, à trouver un boulot avec un vrai bureau, qui m'ont aidé pour tout. Avec le soutien de ces personnes, avec leur intelligence, leur gentillesse, et leur détermination à me crier dessus si le besoin se faisait sentir, même un moins que rien comme moi pouvait avoir la chance de faire quelque chose comme ça.

Katel LeDû, Jeffrey Zeldman et Jason Santa Maria : vous avez construit quelque chose d'incroyable avec A Book Apart, et je suis extrêmement honoré d'avoir pu y tenir mon petit rôle. Je ne saurai jamais vous remercier assez de m'avoir offert cette possibilité.

Peter Richardson et Mike Pennisi, je ne pourrais imaginer deux meilleures personnes pour dissiper le sentiment obsédant d'avoir confondu un terme, oublié un point-virgule ou – horreur des horreurs – pris `undefined` pour une fonction. Je vous en dois une, pour tous les « euh non, en fait... » que je n'entendrai pas après la parution de ce livre.

Erin Kissane, sans ta participation, mon écriture n'aurait jamais mérité de figurer dans ce livre. Quand j'ai su que tu serais mon éditrice, après avoir admiré ton travail pendant tant d'années, je crois que c'est la première fois que j'ai pensé pouvoir y arriver.

À mes collègues de Bocoup et du Filament Group, passés et présents : je n'aurais rien eu à écrire si je n'avais pas eu la chance d'apprendre de JavaScripteurs aussi brillants et responsables que vous. J'espère seulement que vous serez fiers de moi.

LMM, c'est génial que tu aies relu mon bouquin. C'est aussi génial que tu sois ma petite amie. Et c'est encore plus génial que tu le sois restée après m'avoir aidé à travailler sur ce livre. Je n'aurais pas pu le faire sans toi. Je ne pourrais rien faire sans toi. Je te remercie pour beaucoup, beaucoup de choses ; je te remercie pour tout.

Enfin, hé M'man, tu te souviens de cette fois où Papa m'a dit de « publier quelque chose » pour lui, il y a quelques années de ça ? Entre ça et « on devrait réparer un vieux vélo un jour », il continue encore à « m'emmerder ».

M'enfin, un sur deux, c'est déjà pas mal.

RÉFÉRENCES

Les URL abrégées sont numérotées dans l'ordre chronologique ; les URL complètes correspondantes sont listées ci-dessous à titre de référence.

Introduction

00-01 https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

00-02 <https://css-tricks.com/dom/>

Chapitre 1

01-01 <https://abookapart.com/products/responsible-responsive-design>

01-02 https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

Ressources

07-01 <https://developer.mozilla.org/en-US/>

07-02 <https://abookapart.com/products/responsible-responsive-design>

07-03 <https://www.nostarch.com/hemingway>

07-04 <http://shop.oreilly.com/product/9780596806767.do>

INDEX

A

accolades [58](#)
alert() [23](#)
amélioration progressive [121](#), [139](#), [140](#), [142](#)
API [11](#)
API DOM [11](#), [108](#)
arguments [59](#)
ASI [29](#)
async [17](#)

B

booléens [38](#)
Boston Globe [139](#)
boucles [86](#)

- do/while [99](#)
- for [86](#)
- for/in [89](#)
- infinies [103](#)
- while [98](#)

break [44](#), [80](#), [81](#), [82](#), [83](#), [84](#), [101](#), [102](#), [120](#)

C

Chrome [21](#)
clé/valeur [53](#)
commentaires [30](#)
concaténation [37](#), [59](#), [69](#)
console [22](#), [23](#)
continue [101](#)
conversion de type [33](#), [59](#), [60](#), [68](#)

couche

de présentation [9](#), [116](#)

structurelle [9](#)

Croll, Angus [142](#)

D

débogage [21](#)

defer [17](#)

Document Object Model [11](#)

DOM [11](#)

DRY [71](#)

E

éditeur [20](#)

égalité [68](#)

erreurs de syntaxe [23](#)

espacement [30](#)

expressions

conditionnelles [64](#)

contrôle du flux [63](#)

switch [80](#)

F

feuille de styles [9](#), [10](#), [14](#), [15](#), [109](#), [111](#), [116](#), [117](#), [119](#)

Filament Group [139](#), [144](#)

Flickr [11](#)

fonctions [57](#)

G

Geocities [24](#)

grouper des expressions [75](#)

H

`hasOwnProperty` [96](#)
Haverbeke, Marijn [143](#)
héritage prototypal [91](#)
Herman, David [143](#)
HTML5 [15](#)

I

identifiants [43](#)
if/else [64](#)
IIFE [136](#)
indexation [48](#), [87](#)
indices [47](#)
inégalité [72](#)
Internet Explorer [27](#)

J

Jehl, Scott [18](#), [139](#), [142](#)
jQuery [11](#)

L

liste de nœuds [108](#), [113](#), [118](#)
LiveScript [8](#)

M

Mac [21](#)
marathon de Boston [139](#)
Marcotte, Ethan [139](#)
Modernizr [17](#)
moteur JavaScript [9](#)
mots-clés [44](#)
Mozilla Developer Network [142](#)

N

NaN [32](#)

nœud [12](#)

notation

- à crochets [55](#)

- à point [55](#)

- littéral objet [54](#)

null [38](#)

number [33](#)

O

opérateurs

- de comparaison [68](#)

- logiques [74](#)

- NON [74](#)

- relationnels [73](#)

ordre des opérations [34](#)

outils de développement [21](#)

P

parenthèses [34](#)

PC [21](#)

points-virgules [29](#)

portée des variables [45](#)

- globale [45](#), [105](#)

- locale [45](#)

prototype [91](#)

R

refactorisation [72](#)

REPL [22](#), [27](#)

requêtes bloquantes [18](#)

S

script

 chargement [17](#)

 externe [15](#)

 placement [16](#)

sensibilité à la casse [28](#)

Stefanov, Stoyan [143](#)

string [35](#)

T

tableaux [46](#)

types

 de données [32](#)

 d'objets [39](#)

 primitifs [33](#)

U

undefined [37](#)

V

valeurs

 falsy [69](#)

 truthy [69](#)

validation [9](#)

variables [40](#)

W

window [104](#)

À PROPOS DE A BOOK APART

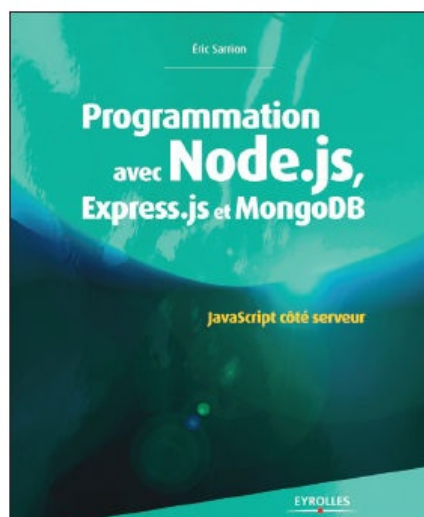
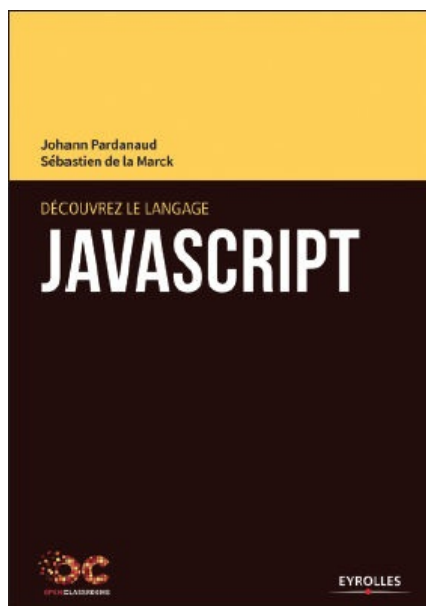
Nous couvrons les sujets émergents et essentiels du design et du développement web avec style, clarté et surtout brièveté – car les professionnels du design et du développement n'ont pas de temps à perdre !

À PROPOS DE L'AUTEUR



Mat « Wilto » Marquis est un web designer professionnel qui passe son temps à maudire sa moto en panne dans les rues de Cambridge. Il est le président du Responsive Issues Community Group, éditeur technique chez *A List Apart*, ex-membre de l'équipe de jQuery, et éditeur de la spécification HTML5. Ce dont Mat est le plus fier, c'est d'avoir terminé *Mega Man 2* en mode difficile, sans perdre une seule vie.

AUX ÉDITIONS EYROLLES



Pour suivre toutes les nouveautés numériques du Groupe Eyrolles, retrouvez-nous sur Twitter et Facebook



[@ebookEyrolles](#)



[EbooksEyrolles](#)

Et retrouvez toutes les nouveautés papier sur



[@Eyrolles](#)



[Eyrolles](#)