

HTML5 APIs & Vanilla Javascript

Support de cours Version 2.0

Sommaire

Sommaire	2
1 Introduction.....	8
2.1 Notions d'architecture applicative	10
2.2 Progressive enhancement ou graceful degradation	12
2 Introduction à HTML 5.....	13
2.1 Anatomie d'une page HTML 5	15
2.2 Démarrer un projet HTML5	16
2.3 Compatibilité des navigateurs.....	18
2.4 Anatomie d'un tableau HTML	19
2.5 Sémantique et structure	21
2.5.1 Nouvelles balises sémantiques.....	21
2.5.2 Microdonnées.....	23
2.5.3 RDFa.....	25
2.5.4 Microformats.....	26
2.5.5 Attributs data-*	28
2.5.6 WAI-ARIA	31
2.6 Formulaires.....	33
2.6.1 Nouveaux types	33
2.6.2 Nouveaux attributs	36
2.6.3 Nouvelles balises	38
2.6.4 Validation	40
2.6.5 Pseudo-classes CSS	41
2.6.6 Rétrocompatibilité et Polyfills	42
2.7 CSS3	50
2.7.1 Mediaqueries et Responsive design.....	50
2.7.2 Sélecteurs CSS3	55
2.7.3 Effets de dégradés	59
3 Bases du langage Javascript	62
3.1 Commentaires	63

3.2 Types de variables "primitifs".....	65
3.2.1 Introduction.....	65
3.2.2 Strings	66
3.2.3 Boolean.....	73
3.2.4 Number et Math.....	74
3.2.5 Dates and Times	86
3.2.6 Null et Undefined	88
3.2.6 Détection de types	91
3.3 Structures conditionnelles.....	93
3.4 Comparaison de contenu et de type	94
3.5 Constantes.....	95
3.6 Tableaux	96
3.6.1 Introduction.....	96
3.6.2 Trier un tableau	100
3.6.3 Tableau multidimensionnel.....	101
3.6.4 Dupliquer des portions de tableaux	102
3.6.5 Recherche dans un tableau	103
3.6.6 Recherche et remplacement	104
3.6.7 Appliquer une fonction à un tableau.....	105
3.6.8 Tableau associatif	107
3.6.9 Méthodes de l'objet Array	109
3.7 Boucles	111
3.7.1 Les boucles while et do	111
3.7.2 La boucle for	112
3.7.3 La boucle for in	114
3.8 Fonctions	115
3.8.1 Introduction.....	115
3.8.2 Fonctions anonymes.....	117
3.8.3 Fonction en exécution immédiate.....	118
3.8.4 Portée des variables (scope)	120

3.8.5 Mode Strict.....	121
3.8.6 Namespaces.....	123
3.8.7 Fonctions en rappel (callback).....	125
3.8.8 Fonctions auto-définissables.....	126
3.8.9 Création de fonction fléchée sous ES6 (« arrow function »).....	128
3.9 Objets	129
3.9.1 Introduction.....	129
3.9.2 Objet personnalisé	129
3.9.3 Prototype.....	133
3.9.4 Constructeurs natifs	134
3.9.5 A quel objet appartient une instance.....	135
3.9.6 Création d'objets sous ES6, avec et sans tranciler.....	136
3.9.7 Sécuriser les objets.....	141
3.9.8 Conclusion	142
3.10 Bonnes pratiques POO	143
3.10.1 Généralités	143
3.10.2 Notion de modularité	147
3.10.3 Data Binding	150
3.11 Templating.....	155
3.11.1 ES6	155
3.11.2 Handlebars.js.....	156
3.11.3 Hyperscript	159
4 Document Object Model (DOM)	161
4.1 Comprendre le DOM	161
4.1.1 Introduction.....	161
4.1.2 Les noeuds.....	163
4.1.3 L'objet Navigator	164
4.1.4 Objets "sub-nodes" hérités de Node.....	167
4.1.5 Propriétés et méthodes pour travailler avec le DOM	169
4.1.6 Identifier le type et le nom d'un noeud	170

4.1.7 Récupérer la valeur d'un noeud	173
4.1.8 Insertion de HTML avec ou sans parsing.....	175
4.1.9 Extraire des éléments de l'arbre DOM	178
4.1.10 Ajouter des noeuds textuels dans le DOM.....	179
4.1.11 Ajout de noeuds avec appendChild() et insertBefore().....	180
4.1.12 Supprimer et remplacer des noeuds dans le DOM	183
4.1.13 Cloner des éléments du DOM	184
4.1.14 Parcourir le DOM.....	186
4.1.15 Vérifier la position d'un noeud.....	189
4.2 Le noeud "document"	189
4.2.1 Introduction.....	189
4.2.2 Déterminer où est le focus	191
4.3 Les noeuds du DOM	191
4.3.1 Element Nodes	191
4.3.2 Sélectionner des noeuds	202
4.3.3 Element Node Style	210
4.3.4 Text Nodes.....	219
4.4 Javascript dans le DOM	223
4.4.1 Insertion et exécution du code Javascript.....	223
4.4.2 Le timing du téléchargement de code.....	224
4.4.3 Recommandations.....	226
4.5 DOM Events.....	227
4.5.1 Introduction.....	227
4.5.2 Référencer la cible d'un évènement	231
4.5.3 Stopper le flow standard avec preventDefault().....	235
4.5.4 Bloquer la propagation d'un évènement	236
4.5.5 Evénement Touch.....	237
4.5.6 Evénement Load, les bonnes pratiques	238
4.5 Les objets Window et Screen	242
4.5.1 Introduction.....	242

4.5.2 Compléments (popup, lookup).....	247
4.5.3 L'objet Screen.....	251
5 Accessibilité (WAI-ARIA).....	252
6 APIs HTML 5.....	258
6.1 QuerySelector.....	258
6.1.1 Rappel et compléments.....	258
6.1.2 Sommer une colonne de tableau	266
6.1.3 Compléments	268
6.2 LocalStorage et SessionStorage.....	269
6.3 Geolocation	274
6.4 Drag & Drop.....	275
6.5 IndexedDB et WebSQL	276
6.6 Canvas.....	277
6.7 SVG	281
6.8 Webaudio	283
6.9 Promise.....	285
6.10 Webworkers	286
6.11 Websockets	286
6.12 Application cache	286
6.13 Deviceorientation et Devicemotion	286
6.14 Touch	287
6.14 RequestAnimationFrame.....	287
6.14 Web Animations.....	288
7 Console, et autres outils.....	289
8 BabelJS et support d'ES6 côté navigateur	301
9 ANNEXE.....	306
9.1 Sélecteurs CSS	306
9.2 Exercices sur les sélecteurs CSS.....	308
9.2.1 Sélecteurs de base.....	310
9.2.2 Sélecteurs descendants.....	314

9.2.3 Sélecteurs sur formulaires.....	317
9.3 La librairie Underscore.JS	320
9.4 Quelques outils à connaître (jslint et jshint)	321
9.5 Extraire les paramètres d'une URL.....	321
9.6 Tracé d'un épicycloïde avec Canvas	323
9.7 Exemple de drag and drop entre listes HTML.....	330
9.x Exemple de templating avec le projet HandleBars.JS.....	331
9.x Tri de liste avec Tablesort	331
9.x Codepen.io et jsbin, les amis du prototypeur	331
9.x La librairie Moment.js.....	331
9.x La librairie Math.js	331
9.x Exemple de fenêtre modale avec Bootstrap	331
10 Bibliographie.....	332
11 Liens utiles, articles	334
11 Changelog	335

1 Introduction

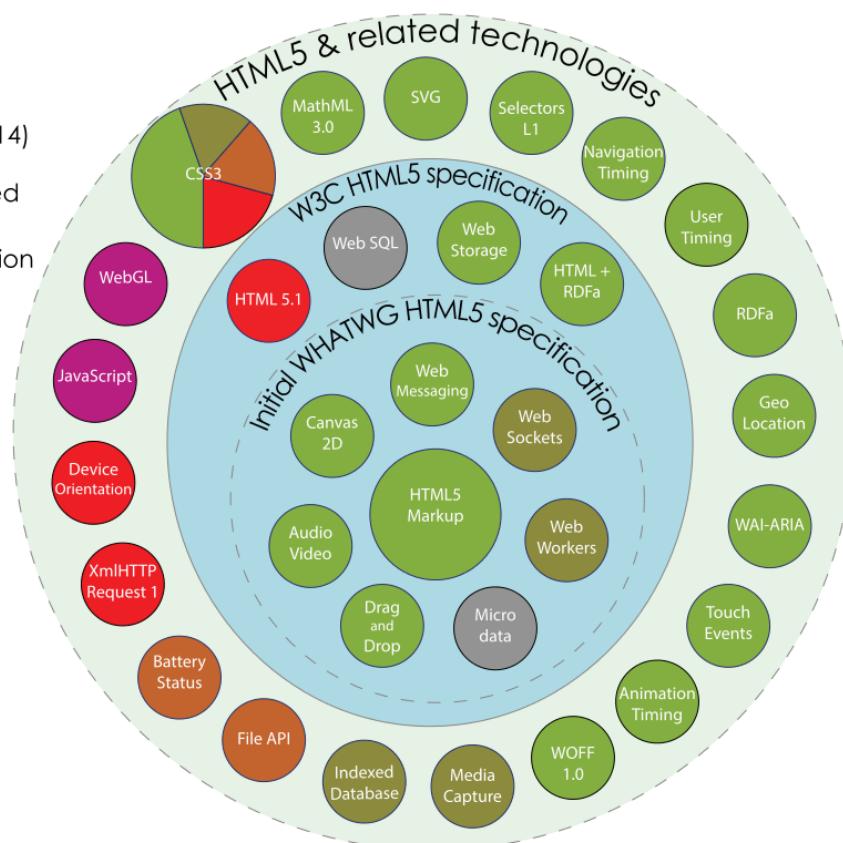
Le langage ECMAScript (plus connu sous le nom de Javascript), et la norme HTML5 sont devenus en quelques années les technologies incontournables pour le développement d'applications webs, qu'elles soient destinées à des postes fixes ou mobiles. Ces technologies sont même devenues incontournables pour tout type d'application, puisqu'elles sont devenues des composants essentiels pour le développement sous Windows, iOS, Android et Linux. Mieux encore, Javascript et HTML5 sont également pertinents pour le développement d'applications de type « desktop » via le projet Electron (développé par Github).

Ce graphique emprunté à Wikipédia (<https://fr.wikipedia.org/wiki/HTML5>) montre la couverture fonctionnelle que propose HTML5, soit au travers de sa spécification, soit au travers de projets parallèles venus se greffer (comme WebGL par exemple).

HTML5

Taxonomy & Status (October 2014)

- Recommendation/Proposed
- Candidate Recommendation
- Last Call
- Working Draft
- Non-W3C Specifications
- Deprecated or inactive



Conçu d'abord comme un support d'auto-formation, avant de devenir un support de cours, ce support est le fruit d'un long travail de recherche, régulièrement mis à jour, du fait des changements nombreux, intervenus tant du côté des normes (en particulier sur Javascript avec ES6) que du côté des pratiques de développement.

Ce support est destiné à des développeurs ayant déjà des bases en développement web, au travers des environnements PHP, Java ou DotNet. Il est vivement recommandé, pour pouvoir suivre ce cours dans de bonnes conditions, de posséder des bases en HTML et en CSS.

Ce cours s'adresse à 3 catégories de développeurs :

- Le développeur débutant en Javascript, qui souhaite acquérir très vite la maîtrise de ce langage, et en acquérir les meilleures pratiques
- Le développeur pratiquant Javascript depuis plusieurs années, qui éprouve le besoin de renforcer ses bases sur certains sujets (par exemple : DOM, Gestion des évènements, Sélecteurs CSS, programmation objet, etc...), ou de prendre connaissance des évolutions apportées au langage lui-même
- Le développeur pratiquant Javascript depuis quelques temps au travers d'un framework et qui éprouve le besoin de renforcer ses connaissances sur les bases de JS, afin de pouvoir trouver plus facilement des solutions palliatives quand le framework ne répond pas pleinement à ses attentes.

Lorsque l'auteur du présent document a commencé à créer la première version de ce support de cours (vers 2010), les navigateurs commençaient tout juste à prendre en compte les caractéristiques de HTML5, et à intégrer les caractéristiques de la norme ECMAScript 5. Pour garantir une bonne compatibilité « cross-browser », il fallait prendre beaucoup de précautions et recourir massivement à des librairies de composants appelés « polyfills » (sur lesquels nous reviendrons). En ce début d'année 2017, la situation s'est aplatie du côté des navigateurs, le développement du marché des terminaux mobiles ayant considérablement accéléré le remplacement des navigateurs. Mais en entreprise, la situation est encore contrastée, et il convient d'être prudent quant aux architectures applicatives, si l'on n'a pas la maîtrise du parc informatique sur lequel une application est destinée à fonctionner.

Entre 2010 et 2017, de nombreux facteurs sont venus bousculer le monde du développement web, et le sortir de la torpeur dans laquelle il était tombé :

- Les frameworks de développement de type SPA (Single Page Application) tels que AngularJS et ReactJS, sont venus bousculer les habitudes, et remettre en question les architectures applicatives
- L'arrivée de NodeJS côté serveur, et de la norme ES6 (progressivement prise en charge par les navigateurs), ainsi que l'arrivée de « transpilers » tels que BabelJS (dont le rôle

est de convertir du code ES6 vers ES5)

- L'apparition de frameworks JS pour les tests unitaires et l'automatisation des tâches
- L'apparition de nombreuses librairies de composants venant combler les lacunes d'ES5 (momentJS, mathJS, underscoreJS, lodash, etc..)
- L'émergence de nouvelles pratiques de développement (« graceful degradation » contre « progressive enhancement »)

2.1 Notions d'architecture applicative

Si vous souhaitez vous concentrer plus rapidement sur les caractéristiques de HTML5 et de JS, vous pouvez allègrement « sauter » ce chapitre, et y revenir plus tard. Mais si vous souhaitez savoir plus précisément comment Javascript et HTML5 se situent dans les différentes architectures applicatives existantes, alors je vous invite à lire la suite de ce chapitre.

On distingue plusieurs types d'architectures applicatives :

- application desktop : application tournant en local sur un PC. Sur ce type d'application, toute installation ou mise à jour nécessite une intervention sur chaque poste, ce qui peut s'avérer compliqué en termes d'administration. Certaines applications desktop nécessitent pour fonctionner de disposer d'une connexion à un serveur hébergeant une base de données centralisée, on parle alors d'application client/serveur, la plupart des applications d'entreprise développées dans les années 90 fonctionnaient sur ce mode. Dans ce contexte, on parle de "client lourd" pour désigner le logiciel installé sur chaque poste. Ce type d'architecture n'a plus le vent en poupe aujourd'hui, on lui préfère des architectures fondées sur les technos web (Javascript, PHP, Python, etc...).

- application web : application composée d'une partie "client" (assemblage de HTML, de CSS et de Javascript) et d'une partie "serveur" (généralement développée en PHP, Python, Ruby, ou encore Javascript avec NodeJS). Le client est dit "léger" car il s'installe dans le navigateur à chaque connexion de l'utilisateur, le déploiement des mises à jour auprès des utilisateurs est du même coup simplifiée. Dans une application web "traditionnelle" (telle qu'on les développait entre les années 2000 et 2010, et encore actuellement pour certaines d'entre elles), c'est le serveur qui pilote quasi intégralement la construction des pages HTML, le rôle de Javascript est souvent réduit à quelques aménagements cosmétiques (notamment via une technique désignée sous le terme "AJAX"), pour améliorer l'ergonomie. Dans ce type d'architecture, l'application ne peut fonctionner si le serveur est temporairement indisponible. Ce type d'architecture est encore très utilisée, mais elle est un peu "datée" et tend de plus en plus à être remplacée par l'architecture SPA.

- application web de type SPA (Single Page Application, littéralement "application fonctionnant sur une seule page web") : c'est une nouvelle approche du développement web, qui s'est amorcée aux alentours de 2009-2010 et qui s'est accentuée depuis lors. Elle consiste à développer la logique métier en Javascript et à l'intégrer directement dans le navigateur, plutôt que de la laisser côté serveur. Certains frameworks de développement Javascript comme BackboneJS, EmberJS, AngularJS, ReactJS et quelques autres sont particulièrement bien adaptés pour ce type de développement. La logique métier se trouvant côté navigateur, la partie serveur (qui peut être écrite en PHP, Python, en Javascript avec NodeJS, en Ruby, etc...) est considérablement réduite : la partie "serveur" sert à "amorcer" l'application, et à envoyer des données actualisées en fonction des demandes du client. En fait, dans ce cas de figure, on peut développer des applications qui stockent dans la mémoire du navigateur l'essentiel des données nécessaires au fonctionnement de l'application. L'application demeure accessible via un navigateur, mais elle devient autonome et est en mesure de fonctionner - au moins temporairement - même si la connexion avec le serveur est coupée (cas d'une coupure réseau par exemple).

De par le fait que la logique métier est concentrée dans le client (le navigateur), les applications de type SPA sont les plus faciles à transformer :

- soit en application desktop via le projet Electron (projet apparu récemment développé par la société Github) : dans ce cas, Electron remplace le navigateur et permet à l'application web de se comporter comme une véritable application desktop
- soit en application mobile hybride via des projets tels que Phonegap (appelé aussi Cordova), Ionic ou Xamarin : ces projets font un peu le même travail qu'Electron, mais cette fois dans le but de packager l'application web pour qu'elle puisse être diffusée comme n'importe application mobile via les stores d'Apple et de Google.

Les logiciels tels que Electron, Phonegap, Ionic ou Xamarin permettent à une application web de dépasser certaines limitations des navigateurs internet. Ils fournissent notamment des APIs permettant de profiter pleinement des différents périphériques du smartphone ou de la tablette (caméra, microphone, GPS, etc..) sur lesquels l'application "tourne".

L'autre catégorie d'application mobile, ce sont les applications mobiles dites "natives". Elles sont développées spécifiquement pour une catégorie de système d'exploitation. Par exemple, une application mobile native pour Android sera développée en Java (avec un environnement de développement fourni par Google), une application développée pour iOS sera développée avec un environnement de développement fourni par Apple (qui utilise le langage Objective C), etc... Cela implique qu'il faut redévelopper une même application plusieurs fois pour tenir compte des spécificités de chaque plateforme. C'est coûteux et risqué, d'où le succès grandissant des applications mobiles hybrides, développées avec un seul code Javascript, et reposant sur des outils comme Phonegap, Ionic ou Xamarin.

2.2 Progressive enhancement ou graceful degradation

Deux principes de développement ont émergé ces dernières années, qui ont fait couler beaucoup d'encre.

- graceful degradation : la "dégradation gracieuse" (ou "dégradation élégante") est une pratique de développement consistant à fournir un certain niveau d'expérience utilisateur sur les navigateurs les plus modernes, tant en garantissant une dégradation élégante vers un mode de fonctionnement généralement moins agréable tout en demeurant fonctionnel, pour les navigateurs plus anciens. Ce niveau inférieur n'est pas aussi agréable à utiliser pour les visiteurs de l'application, mais ils ne se retrouvent pas bloqués et peuvent travailler.
- progressive enhancement : Si le résultat obtenu avec l'"amélioration progressive" est similaire, le processus de développement est "pensé" dans l'autre sens. En résumé, vous commencez par établir un niveau de base pour l'expérience utilisateur, niveau que tous les navigateurs seront en mesure de fournir lors du rendu de votre site Web. Mais vous construisez également des fonctionnalités plus avancées qui seront automatiquement disponibles pour les navigateurs qui peuvent l'utiliser (comme par exemple un mode de saisie de type « drag & drop » disponible uniquement sur les navigateurs qui le supportent).

Ce débat est loin d'être tranché, et on constate sur le terrain que les équipes de développement font souvent l'impasse sur ces 2 principes de développement, cela pour plusieurs raisons :

- Les deux solutions accroissent la complexité des applications, elles accroissent donc aussi la complexité des tests unitaires et fonctionnels
- Ces techniques entraînent des surcoûts de développement, dont le ROI (retour sur investissement) est difficile à justifier auprès des directions générales

Quelques dossiers à lire sur le sujet :

https://www.w3.org/wiki/Graceful_degradation_versus_progressive_enhancement

<https://www.sitepoint.com/progressive-enhancement-graceful-degradation-basics/>

<https://www.sitepoint.com/progressive-enhancement-graceful-degradation-choice/>

2 Introduction à HTML 5



<http://www.w3.org/TR/html5/>

Le W3C (World Wide Web Consortium), est un organisme de normalisation à but non-lucratif, fondé en octobre 1994, chargé de promouvoir la compatibilité des technologies du World Wide Web telles que HTML, XHTML, XML, RDF, SPARQL, CSS, PNG, SVG et SOAP.

Fondé par Tim Berners-Lee au MIT, et actuellement dirigé par lui, le consortium est composé des organisations membres qui mettent à disposition des employés à plein temps dans le but de travailler ensemble à l'élaboration de normes pour le World Wide Web. En date du 12 Avril 2013, le World Wide Web Consortium (W3C) compte 379 membres.

Le W3C est hébergé en Europe par l'INRIA (Institut national de recherche en informatique et en automatique), et il est activement soutenu par l'Union Européenne.



<http://www.whatwg.org>

Le WHATWG (Web Hypertext Application Technology Working Group) est une organisation fondée en 2004 par un groupe d'experts issus de la Fondation Mozilla, d'Opera Software, et d'Apple. Ce groupe dissident, mené par Ian Hickson, fut fondé en réaction à l'orientation prise par le W3C de développer de nouvelles normes pour HTML incompatible avec les anciennes

normes HTML 4 et XHTML 1, et ne répondant pas aux besoins très concrets des webdesigners.

Début 2007, et après l'annonce par le W3C de l'arrêt du projet de ce qui devait être le nouveau HTML, le WHATWG proposa au W3C qu'un nouveau nouveau groupe de travail du W3C prenne en compte les spécifications élaborées par le WHATWG, comme point de départ pour la définition de la nouvelle norme HTML 5. Le W3C accepta la proposition du WHATWG et le démarrage d'un nouveau groupe de travail au sein du W3C fut entériné en mai 2007.

Depuis 2007, le W3C et le WHATWG travaillent en bonne intelligence au développement de la norme HTML 5.

Depuis la mi 2012, les 2 organismes ont convenu d'une nouvelle répartition des tâches :

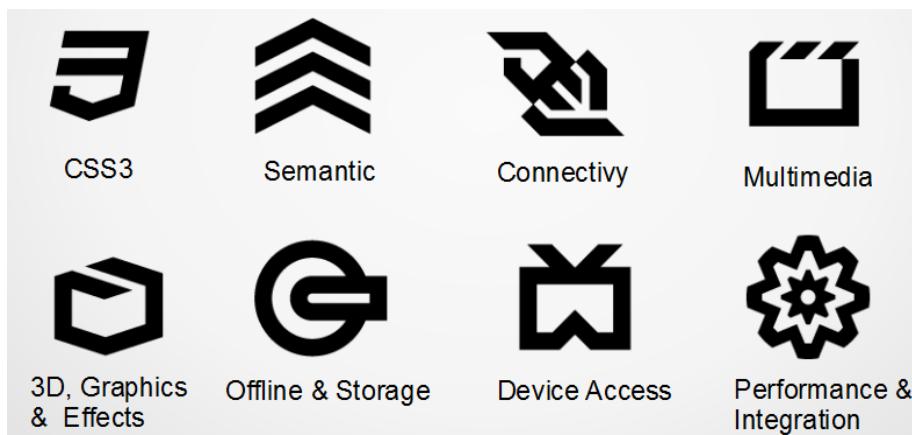
Le W3C s'occupe de la normalisation du HTML 5

Le WHATWG travaille plus particulièrement sur l'évolution de la norme HTML 5, et sur son enrichissement par l'élaboration de nouvelles fonctionnalités (définition de nouvelles balises HTML, définition de nouvelles API, etc...).

L'objectif de ces 2 organisations est le suivant : à terme, on ne parlera plus de HTML 4, de XHTML, de HTML 5, mais d'une seule et même norme, HTML, qui couvrira l'ensemble des spécifications couvertes par HTML 5, et normalisées par le W3C.

Il est important de souligner ici que ce cours constitue un « instantané » d'une norme qui est en cours de rédaction et de stabilisation, et même d'évolution si l'on considère également les travaux en cours au sein du WHAWG. (Cf. le tableau de la « timeline » ci-dessous pris sur <http://en.wikipedia.org/wiki/HTML5>).

Le W3C s'est efforcé de segmenter la spécification en différents chapitres, ou classes, de manière à en faciliter la compréhension. Chacune de ces classes s'est vue attribuer un nom et un logo distinct.



2.1 Anatomie d'une page HTML 5

La structure de l'entête a un peu changé par rapport aux anciennes normes HTML4 et XHTML :

```
<!DOCTYPE html>      Déclaration du DOCTYPE simplifiée en HTML5

<html lang="fr">

  <head>

    <meta charset="utf-8">  charset déclaré avant l'élément "title"

    <title></title>

    <script src="foo.js"></script>  Plus besoin de préciser le type de source

    <link rel="stylesheet" href="foo.css">  Plus besoin de préciser le type de source

  </head>

  <body>

    <div>Ajouter du contenu ici</div>

  </body>

</html>
```

2.2 Démarrer un projet HTML5

Il est possible de démarrer un projet en HTML5 en s'appuyant sur des projets pré-packagés par des experts du sujet, comme le projet "HTML5 Boilerplate" (H5BP), ou encore le projet Bootstrap de Twitter.

En ce qui concerne le projet H5BP, le site officiel est le suivant :

<http://html5boilerplate.com>

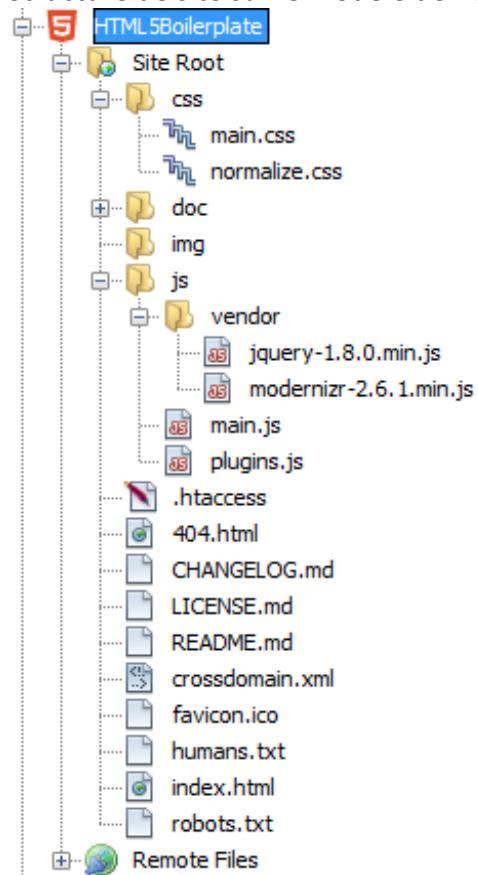
Mais il est préférable de l'aborder au travers du projet Initializr, qui propose plusieurs exemples d'implémentations de H5BP :

<http://www.initializr.com>

Le projet Bootstrap de Twitter est un projet concurrent proposant une approche similaire :

<http://getbootstrap.com>

Structure de site sur le modèle de H5BP :



Structure de la page index.html :

```
<!DOCTYPE html>
<!--[if lt IE 7]>      <html class="no-js lt-ie9 lt-ie8 lt-ie7"> <![endif]-->
<!--[if IE 7]>        <html class="no-js lt-ie9 lt-ie8"> <![endif]-->
<!--[if IE 8]>        <html class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js"> <!--<![endif]-->
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
    <title></title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width">
    <!-- Place favicon.ico and apple-touch-icon.png in the root directory -->
    <link rel="stylesheet" href="css/normalize.css">
    <link rel="stylesheet" href="css/main.css">
    <script src="js/vendor/modernizr-2.6.1.min.js"></script>
</head>
<body>
    <!-- Add your site or application content here -->
    <p>Hello world! This is HTML5 Boilerplate.</p>
    <script src="//ajax.googleapis.com/ajax/libs/jquery/8.0/jquery.min.js">
    </script>
    <script>window.jQuery || document.write(
        '<script src="js/vendor/jquery-8.0.min.js"></script>');
    <script src="js/plugins.js"></script>
    <script src="js/main.js"></script>
    <!-- Google Analytics: change UA-XXXXX-X to be your site's ID. -->
    <script>
        var _gaq=[['_setAccount','UA-XXXXX-X'],['_trackPageview']];
        (function(d,t){var g=d.createElement(t),s=d.getElementsByTagName(t)[0];
        g.src=('https:'==location.protocol)?'//ssl':'//www')+
            '.google-analytics.com/ga.js';
        s.parentNode.insertBefore(g,s)}(document,'script'));
    </script>
</body>
</html>
```

Un commentaire concernant le meta-tag X-UA-Compatible ci-dessous :

```
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
```

Ce meta-tag utilisé avec le paramètre « IE=edge » force IE à utiliser le moteur de rendu le plus récent en sa possession (sachant qu'IE, notamment dans sa version 9, peut être utilisé avec des moteurs de rendu plus anciens (IE 6, 7 ou 8) par défaut).

Le paramètre « chrome=1 » indique à IE que vous lui demandez d'activer le plugin Google Chrome Frame (GCF), s'il est installé sur la version d'IE en cours d'exécution.

Pour de plus amples renseignements sur les paramètres du meta-tag http-equiv :

<http://stackoverflow.com/questions/6771258/whats-the-difference-if-meta-http-equiv-x-ua-compatible-content-ie-edge>

<http://webdesign.about.com/od/metataglibraries/p/x-ua-compatible-meta-tag.htm>

<http://www.google.com/chromeframe?hl=fr>

<http://www.chromium.org/developers/how-tos/chrome-frame-getting-started>

2.3 Compatibilité des navigateurs

Tous les navigateurs ne sont pas égaux face aux nouvelles fonctionnalités proposées par la norme HTML5.

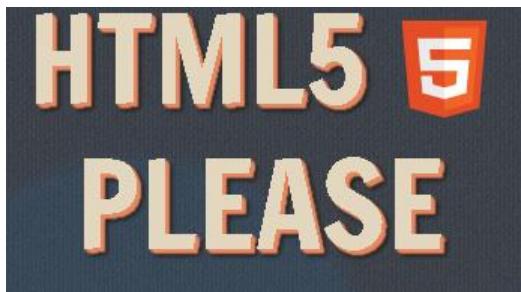
Pour savoir si un navigateur - ou une version de navigateur - supporte une fonctionnalité du HTML5, le site de référence est :

<http://caniuse.com>

The screenshot shows the homepage of caniuse.com. At the top, there is a search bar with the placeholder "border-radius, WebGL, woff, etc". Below the search bar are several navigation links: "Index" (highlighted in orange), "Tables", "Import stats", "FAQ", "Resources", and "Embed". The main content area is divided into three sections: "CSS", "HTML5", and "SVG". Each section contains a list of features with their compatibility status across different browsers. The "CSS" section includes: @font-face Web fonts, calc() as CSS unit value, 2.1 selectors, Counters, and Feature Queries. The "HTML5" section includes: Audio element, Canvas (basic support), Canvas blend modes, Color input type, and contenteditable attribute (basic). The "SVG" section includes: Inline SVG in HTML5, SVG (basic support), SVG effects for HTML, SVG filters, and SVG fonts.

Autre site intéressant proposant des informations complémentaires :

<http://html5please.com>



2.4 Anatomie d'un tableau HTML

La structure des tableaux HTML n'a pas fondamentalement changé avec l'arrivée de HTML5.

On pourrait même considérer que le code des tableaux s'est simplifié, car certaines balises fermantes (comme `</tr>` et `</td>`) sont devenues facultatives en HTML5. Néanmoins, pour assurer une bonne lisibilité et une bonne maintenabilité du code, on recommandera de rester fidèle à une syntaxe de type XHTML, qui est complètement compatible HTML5.

En revanche, un nouvel attribut "scope", est venu renforcer la structure du code HTML. Cet attribut, qui peut prendre les valeurs "col" ou "row" permet de définir la portée de la ligne ou colonne considérée, par rapport aux cellules adjacentes. Cet attribut peut être utile pour la manipulation du tableau en Javascript, mais il apporte de surcroît une information précieuse aux personnes présentant des déficiences visuelles et qui se font aider par des logiciels d'assistance à la lecture (dont on reparlera dans le chapitre "WAI-ARIA").

Exemple de tableau HTML5 utilisant l'attribut "scope" sur certains entêtes de lignes et de colonnes :

```
<!DOCTYPE html>
<html lang="fr">
<head>
<meta charset="utf-8">
<title>Tableau HTML</title>
</head>
<body>
    <table id="ventes"
        summary="Tableau de statistiques des ventes, montants exprimés en M€.">
        <caption>Quarterly Sales*</caption>
        <thead>
            <tr>
                <th scope="col">Sociétés</th>
                <th scope="col">Q1</th>
                <th scope="col">Q2</th>
                <th scope="col">Q3</th>
                <th scope="col">Q4</th>
            </tr>
        </thead>
        <tbody>
            <tr>
                <th scope="row">Société A</th>
                <td>€621</td>
                <td>€942</td>
                <td>€224</td>
                <td>€486</td>
            </tr>
            <tr>
                <th scope="row">Société B</th>
                <td>€147</td>
                <td>€1,325</td>
                <td>€683</td>
                <td>€524</td>
            </tr>
            <tr>
                <th scope="row">Société C</th>
                <td>€135</td>
                <td>€2,342</td>
                <td>€33</td>
                <td>€464</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

2.5 Sémantique et structure

La norme HTML5 apporte avec elle une palette de nouveaux éléments sémantiques, certains sous la forme de nouvelles balises, et d'autres sous la forme d'attributs pouvant s'appliquer aux différentes balises HTML.

2.5.1 Nouvelles balises sémantiques

HTML 5 apporte une série de nouvelles balises qui peuvent être utilisées pour mieux structurer les pages. On ne serait plus obligé dès lors de placer les différents éléments de la page dans des "div" séparées, et de détourner les attributs "id" et "classe" pour y placer des valeurs sémantiques, comme c'était le cas trop souvent jusqu'ici.

Passons en revue ces nouvelles balises de "structure" :

article : destiné à contenir une partie distincte du site, comme un "article" au sens journalistique du terme, ou encore une entrée de blog, ou un descriptif de produit dans le cadre d'une application e-commerce, par exemple.

aside : une portion de page dite "tangentielle", connectée au contenu principal du site, mais considérée comme secondaire et pouvant être placé à côté, au dessus ou en dessous du contenu principal (comme un "sidebar" pour un article de magazine).

nav : zone de navigation d'un document, dans lequel on placera généralement des liens de navigation, voire un menu complet.

section : une section est un regroupement thématique de contenu, comme par exemple le chapitre d'un livre. Une section est généralement destinée à contenir un ou plusieurs "article" (au sens HTML 5 du terme), mais un "article" peut aussi contenir une ou plusieurs "section" (toujours au sens HTML 5 du terme). D'où un certain flou artistique qui fait couler beaucoup d'encre (virtuelle) dans les blogs et forums. Si vous souhaitez utiliser ces balises "section" et "article", vous pourriez définir une règle telle que : "une section peut contenir un ou plusieurs articles, mais un article ne peut contenir aucune section".

footer : une balise attendue depuis fort longtemps, comme la suivante d'ailleurs.

header : même commentaire que pour "footer". Le "header" sera généralement préconisé pour encapsuler les entêtes (cf. balises "h1" à "h6") liés à un article. A noter que l'on peut définir un "header" pour la page, mais aussi un "header" pour chaque "section". Idem d'ailleurs pour "footer".

hgroup : présentée à l'origine comme un moyen de grouper de manière thématique des éléments distincts (comme par exemple des "section" ou "article"), son usage controversé et les dernières informations fournies par le W3C donnent à penser que cette balise pourrait être considérée rapidement comme dépréciée.

Pour illustrer rapidement notre propos, voici comment - avant l'arrivée du HTML5 - les développeurs utilisaient très souvent les "div" pour donner un semblant de structure à leurs pages :

```
<div id="main" class="article">...</div>
```

En HTML5, ils ont la possibilité de remplacer complètement la div par une balise "article", éventuellement en conservant le même "id" s'ils le jugent nécessaire :

```
<article id="main">...</article>
```

Au niveau du CSS3, les nouvelles balises du HTML 5 sont reconnues et on peut dès lors les "styler" sans avoir besoin de recourir à la notion de classe CSS.

Exemple de code combinant plusieurs des nouvelles balises HTML5 :

```
<body>
    <header>en-tête, logo, etc...
        <nav>liens (menu principal)</nav>
    </header>
    <section>Contenu de section
        <article>
            <header>
                <h1>Titre de l'article 1</h1>
                <h2>Sous-titre de l'article 1</h2>
            </header>
            Texte de présentation de l'article 1
            <footer>Infos complémentaires</footer>
        </article>
    </section>
    <aside>Sous-menu vertical ou contenu annexe (news, etc...)</aside>
    <footer>...
        <nav>liens (rappel du menu principal)</nav>
    </footer>
</body>
```

Que faut-il penser de ces nouvelles balises ?

On sent très clairement en observant les noms de ces nouvelles balises qu'elles ont été créées pour satisfaire les demandes des développeurs de blogs, forums, CMS, etc... Dans le cadre du développement d'applications métier, notamment d'applications de gestion, elles présentent un intérêt limité, et cela pour plusieurs raisons :

- elles nécessitent d'apporter d'importantes modifications au code HTML (et donc au CSS) pour un retour sur investissement difficile à évaluer.
- elles ne sont pas encore reconnues de manière homogène par tous les navigateurs, elles posent donc des problèmes de compatibilité avec certains navigateurs anciens. On peut contourner cette difficulté en créant artificiellement ces balises dans le DOM des navigateurs qui en sont dépourvus, mais leur intérêt étant limité, le jeu n'en vaut peut être pas la chandelle.
- elles ne sont pas non plus reconnues de manière homogène par tous les logiciels d'assistance à la lecture (destinés aux personnes présentant des déficiences visuelles et plus généralement aux personnes handicapées)
- si l'intérêt est d'apporter une meilleure structure sémantique au contenu des pages, de nouveaux attributs remplissent parfaitement cet usage, et c'est ce que l'on verra dans les chapitres suivants (cf. les chapitres "microformats", "RDFa", "microdata", "attributs data-*", et surtout "WAI-ARIA").

2.5.2 Microdonnées

Le HTML5 introduit la possibilité de définir une sémantique spécifique dans une page web avec des microdonnées (en anglais : "microdata").

Ces microdonnées permettent d'insérer dans le code HTML des éléments personnalisés - souvent à caractère fonctionnels - mais dans un format facilement exploitable par une machine, au sein d'une page web, à l'aide d'une syntaxe de type "paires clé/valeur".

Ces microdonnées ont d'abord et avant tout vocation à permettre un meilleur référencement des sites internets (certains développeurs de moteurs de recherche se sont beaucoup impliqués dans les groupes de travail du W3C notamment). Mais elles peuvent présenter un intérêt pour les développeurs d'applications "métier" et notamment d'applications de gestion, car elles permettent d'améliorer la structuration du code HTML, et permettent de mettre à disposition

de certains partenaires des pages structurées de manière à ce que ces derniers puissent y récupérer des informations via des robots (une autre manière de proposer du "webservice" en quelque sorte).

Exemple de structuration d'une fiche "utilisateur" dans une application utilisant le principe des microdonnées :

```
<div itemscope itemtype="http://www.my-vocabulary.org/user">
<p>Prénom: <span itemprop="name">Grégory</span>.</p>
<p>Nom: <span itemprop="Lastname">Jarrige</span>.</p>
<p>Photo:  </p>
<p>Adresse: <span itemprop="address">c'est confidentiel, désolé ;)</span>.</p>
<p>Adresse: <span itemprop="address">c'est dans le 91 mais je n'en dirai pas plus</span>.</p>
</div>
```

Pour créer un élément utilisant la syntaxe des microdonnées, vous devez déclarer trois attributs dans les balises HTML standard :

- itemscope : un attribut booléen utilisé pour créer un item ou élément ;
- itemprop : utilisé pour ajouter une propriété à un élément ou un de ses sous-éléments ;
- itemtype : utilisé pour définir un vocabulaire personnalisé.

Pour pouvoir créer un "item", vous devez créer un vocabulaire personnalisé qui permettra de définir la liste des propriétés valides pour les éléments mis à disposition par votre application. Le vocabulaire est une liste de propriétés se présentant sous la forme d'un tableau HTML définissant l'exhaustivité des mots composant un vocabulaire. Vous pouvez bien sûr définir votre vocabulaire, mais si vous ne voulez pas réinventer la roue, vous pouvez aussi utiliser les vocabulaires mis à disposition par les principaux moteurs de recherche, qui collaborent ensemble depuis 2011, pour définir des vocabulaires couvrant de nombreux besoins et secteurs d'activités.

Voici quelques exemples de vocabulaire proposés par le site schema.org :

<http://schema.org/Person>
<http://schema.org/Product>
<http://schema.org/Event>
<http://schema.org/Organization>
<http://schema.org/Review>

De plus, un autre aspect important des microdonnées est lié aux moteurs de recherche. À l'instar de Google, ils sont conçus pour présenter aux utilisateurs les résultats de recherche les plus utiles et les plus informatifs. Les microdonnées n'affectent pas l'aspect du contenu des pages mais elles permettent aux moteurs de recherche de comprendre l'information qui

provient de la page web et de la redistribuer au mieux.

Si vous souhaitez créer votre propre vocabulaire, structurez-le en prenant pour modèle un de ceux qui sont disponibles sur schema.org. Vous pouvez également utiliser un outil mis à disposition par Google. Vous lui donnez une URL valide et il va vérifier si le format de votre page de vocabulaire est conforme à ses attentes :

<http://www.google.com/webmasters/tools/richsnippets>

Les microdonnées constituent l'une des trois méthodes utilisées pour appliquer un marquage "fonctionnel" et structuré au contenu HTML des pages (les deux autres méthodes étant les microformats et les RDF, *Resource Description Framework*).

Les microdonnées disposent d'une API DOM spécifique, au travers de la méthode `getItems()` qui retourne une liste de noeuds (Node List).

Exemple :

```
var items = document.getItems(); // NodeList[]
console.log(items) ;

if (items[0]) {
    var firstItemLen = items[0].properties.length;
    console.log(firstItemLen) ;

    var itemVal = items[0].properties['name'][0].itemValue;
    console.log(itemVal) ;
}
```

2.5.3 RDFA

La RDFA (pour "*Resource Description Format in Attributes*") est une extension du HTML fournissant un jeu d'attributs "parlants".

La syntaxe de base est désignée par le nom de "RDFA Core", et un sous-ensemble simplifié appelé "RDFA Lite" est également disponible. Les deux s'appuient sur un schéma de description de données prédéfini.

Un premier exemple permettra de clarifier le sujet.

Avant l'arrivée de RDFA, si on voulait indiquer qu'un élément HTML correspondait, très souvent on détournait la notion de classe de la façon suivante :

```
<p class="date">2013-04-01</p>
```

Si l'on souhaite s'appuyer sur RDFa Lite, on peut écrire ceci :

```
<p property="http://purl.org/dc/elements/1.1/date">2014-04-01</p>
```

On note que le format de la date présenté ci-dessus est plus lisible pour une machine que pour un humain.

Si l'on souhaite s'appuyer sur RDFa Core, alors on doit écrire ceci :

```
<p property="http://purl.org/dc/elements/1.1/date" content="2014-04-01">1er avril  
2014</p>
```

Grâce à la syntaxe ci-dessus, l'utilisateur "humain" de l'application voit une date dans un format lisible pour lui, mais l'attribut "content" permet de stocker cette même information dans un format exploitable par une machine (un robot de moteur de recherche par exemple).

L'URL indiquée dans l'attribut "property" permet de définir le format que l'attribut "content" doit respecter pour être conforme aux principes de RDFa. En utilisant cette URL, on se trouve redirigé sur le lien suivant, dans lequel on trouve une documentation HTML très détaillée définissant l'élément "date", ainsi que tous les autres types d'éléments référencés par RDFa :

<http://dublincore.org/documents/2012/06/14/dcmi-terms/?v=elements#date>

Certains moteurs de recherche s'appuient sur RDFa pour améliorer le référencement des pages. D'un point de vue du développeur d'une application métier, cette norme apporte un plus pour la sémantique des pages, mais elle peut sembler redondante par rapport aux microdonnées présentées au chapitre précédent. Avant de faire un choix définitif pour une solution ou une autre, il convient d'aborder les microformats.

2.5.4 Microformats

Les microformats ont été créés par une coalition de développeurs désireux de concevoir des patterns de conception de page, s'appuyant sur la norme HTML existante.

Voici un exemple de microformat très simple :

```
<a href="http://flickr.com/photos/tags/kiwi/" rel="tag">A propos des kiwis</a>
```

Les anglophones appellent cela un "Rel-Tag". Cet attribut "rel" est donc un "tag". Cette information permet aux autres machines de savoir que l'URL définie dans ce lien est une page

qui est décrite par un tag, tag dont le nom est défini dans le dernier élément de l'URL, en l'occurrence "kiwi".

Le microformat "hcard" est un autre exemple de microformat, un peu plus complexe (mais pas tant que cela), mais surtout très utile et souvent utilisé, qui définit le format d'une "carte de visite".

Exemple sans microformat :

```
<div class="details">
<p><a href="http://gregphplab.com">Gregory Jarrige</a>
Consultant <a href="http://www.societex.fr">Société X</a>.</p>
</div>
```

Même exemple en version "microformat" :

```
<div class="vcard">
<p><a href="http://gregphplab.com" class="url fn">Gregory Jarrige</a>
Consultant <a href="http://www.societex.fr" class="url org">Société X</a>.
</p>
</div>
```

Contrairement à RDFa et aux microdonnées qui "embarquent" un jeu de nouveaux attributs, les microformats ne "touchent" pas à la norme HTML. On s'appuie en fait sur l'attribut "class", mais on lui attribut une valeur sémantique, avec les valeurs "vcard" (qui définit le contenu des éléments "enfants" de la div), "url fn" (nom du contact) et "url org" (organisation à laquelle le contact est rattaché).

Certains moteurs de recherche, comme Google, utilisent ces microformats pour améliorer leurs résultats de recherche.

Du point de vue du développeur d'applications "métier", l'intérêt d'utiliser les microformats est moins évidente, que pour les microdonnées, ou RDFa. La sémantique apportée par les microformats étant "stockée" dans l'attribut "class" rend l'identification et la récupération des données sous-jacentes moins facile (mais pas impossible) pour des automates.

On peut trouver plus d'informations sur les différents microformats existants en parcourant le site suivant :

<http://microformats.org>

2.5.5 Attributs data-*

La norme HTML 5 apporte une autre manière de donner du sens aux éléments contenus dans une page HTML.

Cette nouvelle manière consiste à ajouter de nouveaux types d'attributs que l'on appelle les attributs "data" (attributs de données). Ce sont des attributs définis librement par le développeur. Ces attributs ne sont pas visibles par l'utilisateur "humain" d'une application, mais ils sont bien facilement identifiables et manipulables avec le langage Javascript, et c'est la raison pour laquelle ils ont la préférence de votre serveur ;).

Exemple sans attribut "data" :

```
<p class="id-443">Produit 443</p>
```

Exemple avec attribut "data" :

```
<p data-id="443">Produit 443</p>
```

Explication :

Avant HTML5, quand on avait besoin de stocker une donnée "métier", comme par exemple un code produit, on détournait très souvent l'attribut "id" (ou quelquefois l'attribut "class") pour y stocker cette information. Dans le cas de l'attribut "id" dont on rappelle que sa contrainte principale est d'être impérativement unique, on ajoutait très souvent un préfixe, comme "id-" dans l'exemple ci-dessus, histoire d'éviter tout conflit avec d'autres éléments de la page.

Avec HTML5, on dispose d'autant d'attributs "data" qu'on le souhaite, il suffit qu'ils commencent par "data-". On est donc tout à fait autorisé à écrire ceci :

```
<p data-id="443" data-SSN="XUL443XXU" data-garantie="3" data-prix=2500.50>Produit  
443</p>
```

Les attributs "data" n'ont pas été créés dans une optique de référencement, mais bien dans l'optique de pouvoir associer des informations fonctionnelles à un élément HTML sans pour autant les faire apparaître sur la page (côté navigateur).

L'arrivée de ce nouveau type d'attribut s'accompagne d'une nouvelle API Javascript permettant de manipuler facilement ces attributs. Cette API se présente sous la forme de 2 méthodes : querySelector() et querySelectorAll() qui répondent à 2 besoins différents. Nous étudierons plus en détail cette API et ces 2 méthodes dans un chapitre dédié, mais voyons quand même ce que l'on peut obtenir à partir de l'exemple précédent :

```
<p data-id="443" data-SSN="XUL443XXU" data-garantie="3" data-prix=2500.50>Produit
```

```
443</p>
```

```
<script>
var el = document.querySelector('p');
console.log('data-id = ' , el.dataset['id']); // data-id = 443
console.log('data-ssn = ' , el.dataset['ssn']); // data-SSN = XUL443XXU
console.log('data-garantie = ' , el.dataset['garantie']); // data-garantie = 3
console.log('data-prix = ' , el.dataset['prix']); // data-prix = 2500.50
</script>
```

On notera que l'attribut data-SSN devient "data-ssn" en Javascript.

On peut également modifier une valeur d'attribut de la façon suivante :

```
el.dataset['id'] = 100;
console.log('Maintenant data-id a pour valeur : ' , el.dataset['id']); // Maintenant
data-id a pour valeur : 100
```

Les développeurs utilisant le framework jQuery peuvent manipuler les attributs "data" en utilisant la méthode data(). Exemple :

```
var id = $(el).data('id'); // La variable "id" reçoit la valeur de l'attribut "data-
id", soit 443
```

On le reverra ultérieurement en approfondissant l'utilisation de la méthode querySelector(), mais il est déjà intéressant de noter que l'on peut sélectionner un attribut "role" ayant une valeur particulière :

```
<div role="main">Lorem ipsum...</div>

<script>
var el = document.querySelector("[role='main']");
console.log(el) ; // <div role="main">
console.log(el.firstChild) ; // <TextNode textContent="Lorem ipsum...">
</script>
```

On peut également sélectionner tous les éléments ayant un attribut "role" défini, et ce quelle que soit sa valeur sur les différents éléments :

```
<div role="main">Lorem ipsum...</div>
<div role="nav">Lorem ipsum...</div>

<script>
var el = document.querySelectorAll("[role]");
console.log(el) ; // NodeList[div, div]
console.log(el[0]) ; // <div role="main">
console.log(el[0].firstChild) ; // <TextNode textContent="Lorem ipsum...">
</script>
```

La méthode `querySelectorAll()` trouve tout son sens dans la sélection d'éléments ayant un même attribut :

```
<p data-id="443" data-SSN="XUL443XXU" data-garantie="3" data-prix=2500.50>Produit  
443</p>  
<p data-id="442" data-SSN="XUL443XXV" data-garantie="2" data-prix=1500.50>Produit  
442</p>  
<p data-id="444" data-SSN="XUL443XXT" data-garantie="1" data-prix=500.50>Produit  
444</p>  
<p data-id="445" data-SSN="XUL443XXZ" data-garantie="3" data-prix=800.50>Produit  
445</p>  
  
<script>  
var el = document.querySelectorAll('[data-id]');  
console.log(el);  
console.log('data-id = ' , el[0].dataset['id']); // data-id = 443  
console.log('data-ssn = ' , el[1].dataset['ssn']); // data-SSN = XUL443XXV  
console.log('data-garantie = ' , el[2].dataset['garantie']); // data-garantie = 1  
console.log('data-prix = ' , el[3].dataset['prix']); // data-prix = 800.50  
</script>
```

Nous verrons d'autres exemples d'utilisation des attributs `data` au cours de la formation, ainsi que d'autres utilisations des méthodes `querySelector()` et `querySelectorAll()`.

Il peut arriver que l'on ait besoin de récupérer un « dataset », soit la liste des attributs « `data` » affectés à un nœud du DOM, et que l'on souhaite itérer sur cette liste. Or un « `dataset` » est un objet de type `DOMStringMap`, pour lequel il n'existe pas d'itérateur standard. On peut contourner la difficulté en utilisant l'exemple suivant :

```
var dataset_items = monobjet.dataset ; // monobjet contient un dataset de x items  
var dataset_keys = Object.getOwnPropertyNames(dataset_items);  
var dataset_length = dataset_keys.length ;  
console.log(dataset_items) ;  
console.log(dataset_length) ;  
console.log(dataset_keys) ;  
var i = 0 ;  
for (i = 0 ; i < dataset_length ; i+=1) {  
    console.log(dataset_keys[i] + ' = ' + dataset_items[dataset_keys[i]]);  
}
```

Les attributs `data` sont plébiscités -à juste titre - par les développeurs de sites très connus tels que Twitter, Facebook, Google et LinkedIn. Je vous encourage à les utiliser dans vos propres applications.

2.5.6 WAI-ARIA

WAI-ARIA est l'acronyme de "Web Accessibility Initiative's Accessible Rich Internet Applications"

Un groupe de travail au sein du W3C travaille activement sur la problématique de l'accessibilité des applications, pour les personnes victimes de handicaps (visuels ou moteurs). Ces personnes ont besoin pour travailler sur des applications de l'assistance de logiciels de lecture d'écran comme JAWS.

WAI-ARIA apporte une solution à cette problématique avec l'arrivée de nouveaux attributs HTML permettant aux logiciels d'assistance à la lecture de réagir de manière plus pertinente à certains types d'évènements.

Par exemple, un lien tel que celui ci-dessous apporte peu d'information aux logiciels de lecture sur le type d'action effectué :

```
<a href="http://example.com">Lancer un popup</a>
```

Avec un lien tel que celui ci-dessus, le logiciel de lecture ne peut pas donner d'information contextuelle pertinente à l'utilisateur. WAI-ARIA introduit un nouvel attribut, "aria-haspopup" qui va donner à l'utilisateur une information quand au type d'action déclenché par ce lien. Exemple :

```
<a href="http://example.com" aria-haspopup="true">Lancer un popup</a>
```

Un nombre important de nouveaux attributs sont apparus en HTML 5 pour répondre à ce type de problématique. Certains de ces attributs sont des rôles définissant le type d'usage que l'on peut faire de certains éléments HTML. D'autres attributs correspondent à une notion de statut, apportant des informations sur l'état de certains éléments à un instant donné.

Ces nouveaux attributs sont plus simples à mettre en oeuvre que les nouvelles balises HTML 5. Ils sont d'ailleurs mieux supportés par les navigateurs et les logiciels de lecture que les nouvelles balises précitées.

Le nouvel attribut "role" s'accompagne d'une série de valeurs prédéfinies, qui ne correspondent pas exactement aux nouvelles balises structurelles du HTML5, mais ce n'est pas un problème. Par exemple, on peut définir une bannière de la façon suivante :

```
<div role="banner">...</div>
```

Le rôle "banner" est analogue à la balise "header" d'un point de vue sémantique, mais il offre

l'avantage de ne pas remettre en cause la structure de la page.

Autre exemple de rôle :

```
<div role="main">...</div>
```

Huit valeurs sont définies pour l'attribut "role" dans la spécification WAI-ARIA :

- **application** : Affiche une zone de page correspondant à une application interactive et non à un document
- **banner** : contenu de site placé dans l'entête ou le pied de page, analogue à la balise "header"
- **complementary** : contenu généralement placé dans un "sidebar", et rattaché à une autre partie de la page ayant pour rôle "main". Peut être considéré comme analogue à la balise "aside"
- **contentinfo** : portion de page fournissant des informations - juridiques ou autres - concernnant le document consulté. Peut être considéré comme étant analogue à la balise "footer".
- **form** : permet d'indiquer la présence de tout type de formulaire, à l'exception des formulaires de recherche
- **main** : signale le contenu principal de la page
- **navigation** : contient des groupes de liens, ou un menu. Peut être considéré comme étant analogue à la nouvelle balise "nav".
- **search** : indique la présence d'un formulaire de recherche.

On peut utiliser les méthodes `querySelector()` et `querySelectorAll()` pour sélectionner les éléments utilisant un attribut particulier, comme dans l'exemple suivant :

```
<div role="main">Lorem ipsum...</div>

<script>
var el = document.querySelector("[role='main']");
console.log(el); // <div role="main">
</script>
```

On peut aussi "styler" très facilement des éléments possédant un attribut WAI-ARIA :

```
<style>
div[role='main'] {
    background-color: blue;
}
</style>
```

Ce chapitre était une courte introduction aux notions d'accessibilité et à la spécification WAI-ARIA. On reviendra plus en détail sur le sujet dans un prochain chapitre.

2.6 Formulaires

2.6.1 Nouveaux types

La norme HTML5 fournit plusieurs nouveaux types de champs de formulaires.

Ces nouveaux types ne sont pas adoptés de manière homogène par tous les navigateurs, mais ce n'est pas dramatique car les navigateurs ont pour particularité de traiter comme des champs de type "text" ceux des nouveaux types qu'ils ne reconnaissent pas encore.

Parmi les nouveaux types de champ, on trouve :

```
<input type="search"> pour la saisie dans une "boîte" de recherche
<input type="email"> pour la saisie d'email
<input type="url"> pour la saisie d'URL
<input type="tel"> pour la saisie de numéro de téléphone
<input type="number"> pour la saisie de nombre
<input type="range"> pour la saisie de donnée au moyen d'un curseur ("slider")
<input type="date"> pour la saisie de date
<input type="month"> pour la saisie de mois
<input type="week"> pour la saisie de semaines
<input type="time"> pour la saisie d'heure/minutes/seconde
<input type="datetime"> pour la saisie de timestamps
<input type="datetime-local"> pour la saisie de timestamps en temps absolu (UTC)
<input type="color"> pour des effets de type "colorpicker"
```

Certains navigateurs appliquent un style par défaut à ces nouveaux types de champ.

Mais surtout, certains terminaux mobiles personnalisent automatiquement l'affichage de la saisie en fonction du type de champ (en proposant par exemple un pavé numérique sur les champs de type "number").

Les types "number" et "range"

Voici les nouveaux types de champ "number" et "range" et leur affichage sur Google Chrome :

```
<input type="number"><br>
<input type="range"><br>
```



Les champs de type "input" bénéficient d'attributs particuliers permettant de les personnaliser :

```
<input type="number" max="100" min="10" step="10" value="50">
```

On peut aussi manipuler l'attribut "step" au moyen des méthodes `stepUp()` et `stepDown()`, de la

façon suivante :

```
var foo = document.querySelector('input[type=range]');
foo.stepDown(3);
```

On peut également récupérer la valeur d'un champ au moyen de la propriété valueAsNumber :

```
var foo = document.querySelector('input[type=number]');
bar = foo.valueAsNumber;
console.log(bar);
```

Les types "date", "time", "datetime", "month", etc...

HTML5 apporte de nouveaux champs permettant la saisie de données temporelles.

Exemple de champ de type "date" et sa représentation dans Google Chrome :

```
<input type="date">
```



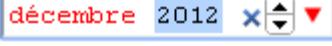
Exemple de champ de type "time" et sa représentation dans Google Chrome :

```
<input type="time">
```



Exemple de champ de type "month" et sa représentation dans Google Chrome :

```
<input type="month" max="2016-06" min="2012-01" step="3">
```



On notera que les méthodes stepUp() et stepDown() fonctionnent aussi sur ce type de champ.

Exemple de champ de type "datetime-local" et sa représentation dans Google Chrome :

```
<input type="datetime-local">
```



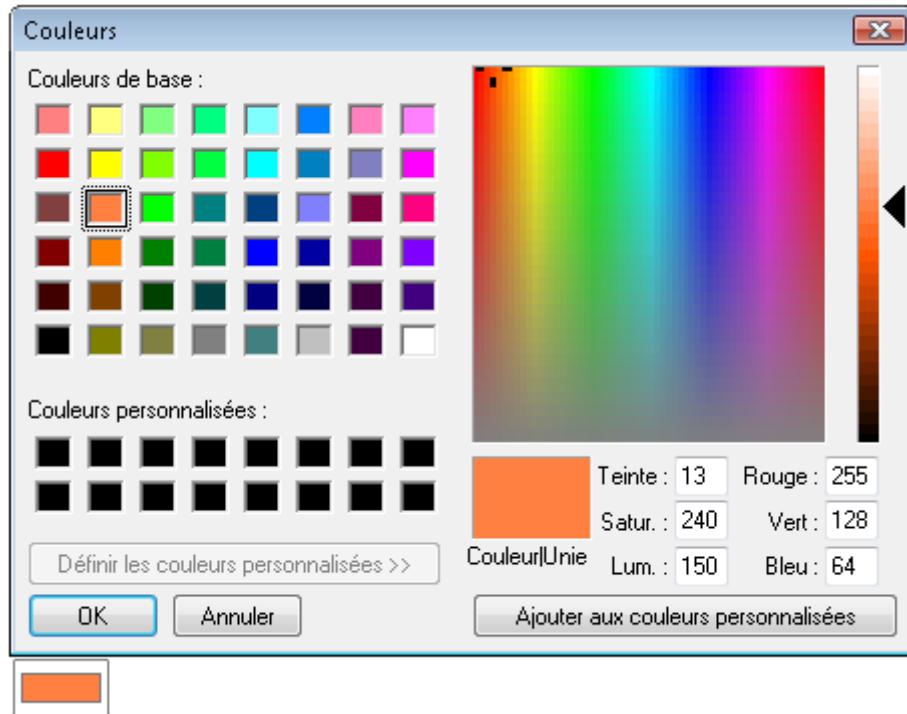
On peut récupérer la valeur d'un champ de type via un sélecteur Javascript associé à la méthode valueAsDate() :

```
var foo = document.querySelector('input[type=date]');
bar = foo.valueAsDate;
```

Le type "Color"

Exemple de champ de type "color" et sa représentation dans Google Chrome :

```
<input type="color">
```



Discussion : Le support des nouveaux types de champ n'est pas homogène sur les différents navigateurs du marché. Cela peut poser des problèmes, et on verra au chapitre "Rétrocompatibilité et Polyfills" qu'il existe des solutions de contournement pour pallier les lacunes de certains navigateurs.

Mais ce n'est pas sur le terrain des nouveaux types de champs que HTML5 est le plus intéressant, c'est plutôt sur le terrain des nouveaux attributs, et c'est ce que nous allons voir au chapitre suivant.

2.6.2 Nouveaux attributs

La norme HTML5 apporte de nouveaux attributs pour les champs de formulaires.

autofocus

L'attribut autofocus peut être utilisé sur tous les types de champ saisie, et il permet - comme son nom l'indique - de placer le focus automatiquement sur le champ sur lequel cet attribut est défini.

Exemple :

```
<input type="text" autofocus>
```

placeholder

L'attribut placeholder a été popularisé par certains frameworks et il fait maintenant partie intégrante de la norme HTML5.

Exemple :

```
<input type="email" placeholder="e.g. foo@bar.com">
```

autocomplete

Par défaut, la plupart des navigateurs stockent dans un cache au minimum la dernière saisie effectuée sur un champ de formulaire.

Ce comportement peut être gênant sur certaines applications, aussi on peut maintenant le bloquer via l'attribut autocomplete.

Exemple :

```
<input type="email" autocomplete="off">
```

L'attribut "autocomplete" peut aussi être placé sur la balise "form", de manière à impacter tous les champs du formulaire en question.

spellcheck

Certains navigateurs offrent une fonction de vérification orthographique, essentiellement sur les champs de type textarea.

On peut maintenant forcer les navigateurs à appliquer cette fonctionnalité sur d'autres types de champs, via l'attribut spellcheck.

Exemple :

```
<input type="text" spellcheck="true">
```

Le dictionnaire orthographique utilisé est celui correspondant à la langue par défaut du navigateur, mais on peut modifier ce comportement via l'attribut lang.

Exemple :

```
<input type="text" spellcheck lang="fr">
```

multiple

L'attribut multiple est principalement utilisé avec le champ de saisie de type "file" pour

permettre la saisie de plusieurs fichiers. On peut aussi l'utiliser avec le type de champ email, pour la saisie de plusieurs adresses email :

```
<input type="file" multiple>
```

form

L'attribut "form" associé à un champ de formulaire permet de lier explicitement un champ de saisie à un formulaire. Il est dès lors possible de placer ce champ de saisie n'importe où dans la page, et surtout en dehors des balises `<form>` et `</form>` délimitant le formulaire en question.

```
<form id="foo">...</form>
<input type="text" id="bar" form="foo">
```

required

C'est certainement l'un des nouveaux attributs les plus intéressants. Il permet de rendre un champ de sa saisie obligatoire, sans avoir besoin de recourir à du code Javascript pour ce faire :

```
<input type="email" required>
```

pattern

L'attribut pattern permet d'appliquer une expression régulière (ou regex) pour contrôler la validité d'une saisie.

Par exemple : pour forcer le navigateur à n'autoriser que la saisie de valeurs numériques dans un champ, on peut utiliser le pattern ci-dessous :

```
<input type="tel" pattern="\d*>
```

On peut personnaliser le message d'erreur associé au pattern, en ajoutant un second attribut, qui est l'attribut "title" :

```
<input type="tel" pattern="\d*" title="Numbers only">
```

novalidate

On peut désactiver complètement la validation d'un formulaire au moyen de l'attribut novalidate :

```
<form action="foo" novalidate>...</form>
```

On peut aussi appliquer ce principe à des champs de saisieAnd you can do this at a more local level with the formnovalidate attribute

on an input or button element.

```
<button type="submit" formnovalidate>Go</button>
```

contenteditable

Ce n'est pas véritablement un attribut destiné aux formulaires. Il serait plutôt adapté pour rendre "éditable" des cellules de tableaux, ou des paragraphes à l'intérieur d'une page. Son utilisation nécessite l'utilisation de code Javascript, pour pouvoir prendre en compte les modifications de valeur, c'est pourquoi nous reverrons cet attribut ultérieurement.

```
<td contenteditable="true">200</td>
```

Petite astuce Javascript : on peut rendre l'intégralité d'une page éditable en utilisant le code ci-dessous

```
document.body.contentEditable=true
```

AVERTISSEMENTS :

Il convient de rappeler que des attributs de validation tels que "required" et "pattern" ne peuvent en aucun cas constituer des outils suffisants pour contrôler et valider la qualité des données saisies par les utilisateurs. Pour des raisons de sécurité, ces contrôles doivent impérativement être "doublés" par des contrôles - au moins équivalents - effectués côté serveur.

2.6.3 Nouvelles balises

La norme HTML5 propose plusieurs nouvelles balises intéressantes, mais dont le support par les navigateurs est relativement hétérogène.

datalist

La balise `<datalist>` spécifie une liste d'options prédéfinies pour un élément de formulaire de type `<input>`. Elle fournit un mécanisme d'auto-complétion. Pour associer une « `datalist` » à un « `input` », on utilise l'attribut « `list` » au niveau de l' « `input` », attribut dans lequel on indique l'ID de la « `datalist` », comme dans l'exemple ci-dessous :

```
<label for="url_field">Saisissez ou sélectionnez une URL</label>
<input type="url" id="url_field" name="url_field" list="urls"
placeholder="http://www.votresite.fr" />
<datalist id="urls">
  <option label="W3C" value="http://www.w3.org/">
  <option label="WHATWG" value="http://www.whatwg.org/">
  <option label="HTML 5 Specification" value="http://www.w3.org/TR/html5/">
  <option label="SVG Specification" value="http://www.w3.org/TR/SVG/">
  <option label="SPARQL" value="http://fr.wikipedia.org/wiki/SPARQL">
  <option label="UNIX Specification, Version 3" value="http://www.unix-
systems.org/version3/">
</datalist>
```

Attention : la balise « `datalist` » est séduisante, mais la spécification du W3C était floue sur certains points, les éditeurs de navigateurs ne l'ont pas tous implémentée de la même façon. Par exemple, certains navigateurs réalisent l'auto-complétion avec une recherche de type « contient », alors que d'autres navigateurs effectuent une recherche de type « commence par ». De plus, le rendu de l'attribut facultatif « `label` » n'est pas le même selon les navigateurs. Ce flou artistique rend l'utilisation de la balise « `datalist` » problématique, et il apparaît encore

difficile de se passer des fonctions d'auto-complétion proposées par jQuery et quelques autres frameworks Javascript.

progress

Les barres de progression sont couramment utilisées dans les applications, qu'elles soient "web" ou pas.

```
<progress max="20" min="10" value="15">15</progress>
```

Balise Progress



On peut modifier - via Javascript - la valeur d'une barre de progression, au moyen du code suivant :

```
var progressBar = document.querySelector('progress'),  
updateProgress = function (newValue) {  
    progressBar.value = newValue;  
};
```

meter

La balise "meter" se rapproche beaucoup, dans son mode de fonctionnement, de la balise "progress". Elle est plus particulièrement destinée à effectuer des mesures pour faciliter la production de tableaux de bord.

Exemple :

```
<p>  
    Balises meter 1  
    <meter min="35" max="43" value="37.5">37.5°C</meter>  
</p>  
<p>  
    Balise meter 2  
    <meter low="0" high="4" max="5" value="4.5" optimum="2.50">4.5 m</meter>  
</p>
```

Balises meter 1



Balise meter 2



output

L'élément "output" affiche le résultat d'un calcul, éventuellement en interaction avec des saisies utilisateurs.

Exemple :

```
<label for="output">Output</label>
<input type="range" id="range">
<output id="output" for="range"></output>

<script>
    var range = document.getElementById('range'),
        output = document.getElementById('output');
    output.defaultValue = 50;
    range.addEventListener('change', function(e) {
        var newValue = e.currentTarget.value;
        output.value = newValue;
    });
</script>
```

Output  21

2.6.4 Validation

Le moyen le plus simple de valider un élément de formulaire est d'utiliser la méthode `checkValidity()`.

Exemple avec un champ de type "email" :

```
<input type="email">

<script>
    var email = document.getElementById('email');
    function checkStatus(e) {
        var valid = e.currentTarget.checkValidity();
        var validMsg = e.currentTarget.nextSibling;
        var status = (valid) ? 'P' : 'O';
        validMsg.textContent = status;
    }
    email.addEventListener('input', checkStatus, false);
</script>
```

On peut attacher à notre champ de saisie un évènement de type "invalid", ce qui permet de déclencher des actions particulières - au travers de la fonction callback `resaisieEmail()` - dans le cas où le contenu du champ est invalide :

```
email.addEventListener('invalid', resaisieEmail, false);
```

Exemple :

```
<input type="tel" id="tel-home">
<input type="tel" id="tel-work">
<input type="submit">

<script>
var telHome = document.getElementById('tel-home'),
telWork = document.getElementById('tel-work');
telWork.addEventListener('change', function (e) {
    if (e.currentTarget.value === telHome.value) {
        telWork.setCustomValidity('Doivent être différents');
        console.log('erreur');
    } else {
        telWork.setCustomValidity('');
        console.log('good');
    }
}, false);
</script>
```

2.6.5 Pseudo-classes CSS

Une nouvelle catégorie de pseudo-classes CSS a été introduite en HTML5. Elle ont pour but de faciliter la personnalisation de l'affichage, notamment pour les éléments de formulaires.

Par exemple, les champs de formulaires obligatoires (attribut "required") peuvent être "stylés" au moyen de la pseudo-classe :required.

Et les champs optionnels peuvent être "stylés" au moyen de la pseudo-classe :optional.

Exemple :

```
<style>
    input:required { border-color: red; }
    input:optional { border-color: silver; }
</style>
```

Les pseudo-classes :valid et :invalid sont utiles pour appliquer un style particulier aux champs dont le contenu est valide, et à ceux dont le contenu est invalide.

```
<style>
    input:valid { color: green; }
    input:invalid { color: red; }
</style>
```

On peut "styler" les éléments de formulaire selon un attribut particulier. Par exemple, les champs ayant l'attribut "disabled" peuvent être "stylés" avec la pseudo-classe :disabled. L'inverse est vrai pour l'attribut "enabled". Pour un champ de formulaire "readonly", vous

disposez de la pseudo-classe :read-only (l'inverse de cette pseudo-classe étant :read-write).

Les boutons radios et cases à cocher bénéficient de pseudo-classes dédiées, comme :indeterminate quand aucune case n'est cochée, et surtout :checked pour attribuer un style particulier à des case cochées.

Curieusement, il n'existe pas de pseudo-classes pour "styler" les cases non cochées, mais on peut contourner la difficulté en utilisant le sélecteur suivant :

```
var notchecked = document.querySelectorAll("input[type='checkbox']:not(:checked)") ;
```

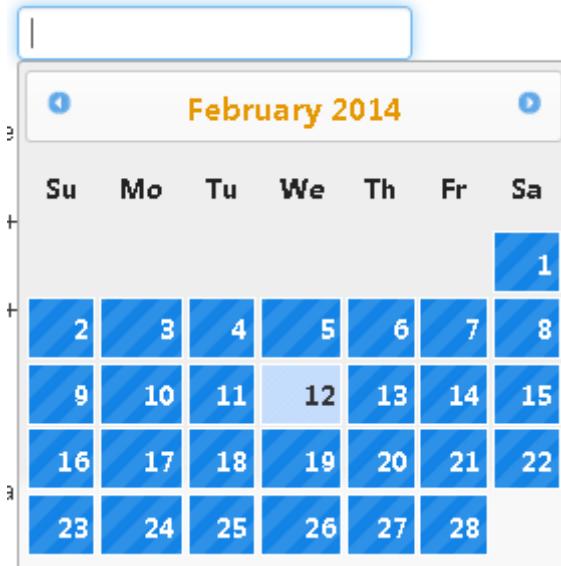
Pour les champs de type "number" et "range", on dispose des pseudo-classes :in-range et :out-of-range, qui se passent de commentaires.

2.6.6 Rétrocompatibilité et Polyfills

Les nouveaux types de champs présentés au chapitre précédent sont intéressants, mais ils déçoivent un peu, et ce pour plusieurs raisons :

- tous les navigateurs ne les "traitent" pas de manière homogène, ce qui pose des problèmes de présentation et surtout de comportement, pour les applications qui les utilisent, et pour leurs utilisateurs bien évidemment,
- les champs de type date, time, etc... ne permettent pas une forte personnalisation, comme le permettent les widgets fournis par certains frameworks (comme jQueryUI, YahooUI, Dojo, etc...) qui fournissent souvent plusieurs types de calendriers de saisie.

Exemple de widget de saisie de type calendrier, intégré au framework jQueryUI (fonctionne et apparaît de manière homogène sur l'ensemble des navigateurs) :



Pour contrôler si un des nouveaux types de champ est supporté par un navigateur, la solution passe par Javascript et se déroule en 3 étapes:

1 - créer dans la mémoire du navigateur un élément <input>

```
var i = document.createElement("input");
```

2 - ajouter au nouvel élément un attribut correspondant au type de champ à détecter, par exemple pour un champ de type "datetime" :

```
i.setAttribute("type", "datetime");
```

3 - dernière étape, vérifier que le type du champ créé virtuellement n'est pas de type "text", ce qui confirmera que le type "datetime" a été retenu :

```
return i.type !== "text";
```

Explication : si le navigateur supporte le type de champ "datetime", il va conserver cette propriété, sinon il va la rejeter au profit de la propriété standard (text). Pour savoir si le type "datetime" a été conservé, il suffit de vérifier que le type du champ est bien différent de "text".

Dans le cas l'où on souhaite vérifier si un type de champ de saisie supporte un attribut particulier, comme par exemple l'attribut "placeholder", on peut utiliser une fonction telle que celle ci-dessous :

```
function supports_input_placeholder() {
```

```

        var i = document.createElement('input');
        return 'placeholder' in i;
    }
}

```

ou une fonction plus générique telle que :

```

function supports_input_attribute(attrName) {
    var i = document.createElement('input');
    return attrName in i;
}

```

Les techniques décrites ci-dessus sont à la base des "polyfills" (ou "polyfillers"), que l'on peut traduire en « prothèse d'émulation ». Mais qu'est ce qu'un "polyfill" ? Pour l'expliquer, revenons un peu en arrière.

En 2002, un développeur du nom de Sjoerd Visscher découvrit qu'il était possible d'intégrer dans le DOM d'IE des éléments qui ne faisaient pas partie du standard HTML, en utilisant la technique suivante :

```
document.createElement('header');
```

A partir de là, l'élément existe dans la mémoire du navigateur, mais c'est un élément virtuel qui n'appartient pas au DOM, et n'apparaît pas dans la page du navigateur. Il nous est néanmoins utile pour forcer la prise en compte du CSS sur les éléments de même nom qui seront eux embarqués dans le code HTML. Si l'on avait souhaité ajouter cet élément dans la page, on aurait dû ajouter le code suivant :

```

var header_field = document.createElement('header');

document.body.appendChild(header_field);

```

Il faut souligner que la méthode createElement() doit être exécutée au tout début du chargement de la page, pour que le CSS puisse s'appliquer aux éléments concernés. Le code doit donc être placé le plus tôt possible dans la page.

A partir du moment où un élément est créé sur ce principe, il est recommandé de lui appliquer au moins un style par défaut, comme dans l'exemple suivant :

```

<style>

    article, aside, details, figcaption, figure, footer, header, hgroup, menu, nav,
    section { display: block; }

</style>

```

En 2008, lors d'un échange sur un forum, Sjoerd Visscher communiqua sa découverte, qui suscita l'enthousiasme de plusieurs experts (notamment Ian Hickson, leader du WHATWG, et John Resig, leader du projet jQuery), car elle sous-entendait la possibilité d'écrire une librairie qui serait une sorte de cale (de « shiv »), permettant de contourner les limites imposées par les vieilles versions d'IE. C'est Remy Sharp qui prit le leadership du projet, soutenu par plusieurs experts Javascript. Le projet prit curieusement le double nom de « HTML 5 shiv » et « HTML 5 shim ». Le code source est téléchargeable via le lien suivant :

<https://code.google.com/p/html5shiv/>

On peut l'utiliser en local après l'avoir téléchargé, ou l'utiliser « en ligne », au moyen du code suivant (à placer au tout début du bloc « head » de la page) :

```
<!--[if lt IE 9]>
<script src= "http://html5shiv.googlecode.com/svn/trunk/html5.js">
</script>
<![endif]-->
```

Pour un historique complet de cette librairie, suivez ce lien :

<http://www.paulirish.com/2011/the-history-of-the-html5-shiv/>

La librairie HTML 5 Shiv a été incorporée à un autre projet open source, Modernizr. En plus de détecter les lacunes des navigateurs, Modernizr apporte des fonctionnalités supplémentaires sur lesquelles nous allons revenir. Il est téléchargeable ici :

<http://modernizr.com>

Download Modernizr 2.7.1

Use the [Development version](#) to develop with and learn from. Then, when you're ready for production, use the build tool below to pick only the tests you need.

CSS3

TOGGLE

- @font-face
- background-size
- border-image
- border-radius
- box-shadow
- Flexible Box Model (flexbox)
- Flexbox Legacy
- hsla()
- multiple backgrounds
- opacity
- rgba()
- text-shadow
- CSS Animations
- CSS Columns
- CSS Generated Content (before/after)
- CSS Gradients
- CSS Reflections
- CSS 2D Transforms
- CSS 3D Transforms
- CSS Transitions

HTML5

TOGGLE

- applicationCache
- Canvas
- Canvas Text
- Drag & Drop
- hashchange
- History (pushState)
- HTML5 Audio
- HTML5 Video
- IndexedDB
- Input Attributes
- Note: does not add classes
- Input Types
- Note: does not add classes
- localStorage
- postMessage
- sessionStorage
- Web Sockets
- Web SQL Database
- Web Workers

Misc.

TOGGLE

- Geolocation API
- Inline SVG
- SMIL
- SVG
- SVG clip paths
- Touch Events
- WebGL

Extensibility

Non-core detects

GENERATE!

DOWNLOAD

Custom Build

Incorporer le code de Modernizr au tout début de vos pages :

```
<script src="js/vendor/modernizr-2.7.1.min.js"></script>
```

Premier exemple d'utilisation (détection du support de la balise « canvas ») :

```
<script>
  if (Modernizr.canvas) {
    alert("Ce navigateur supporte HTML5 canvas!");
  } else {
    alert("Ce navigateur ne supporte pas HTML5 canvas!");
  }
</script>
```

Les « polyfills » sont des librairies de fonctions Javascript destinées à combler les lacunes en HTML 5, des navigateurs trop anciens.

« Combler les lacunes » se traduit en anglais par « fill in the gaps » => « polyfills »...

On peut utiliser Modernizr pour « charger » dynamiquement certains « polyfills ».

On trouve un classement des « polyfills » pouvant être utilisés avec Modernizr, sur le site suivant :

<https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-browser-Polyfills>

Exemple 1 : Chargement d'un « polyfill » destiné à pallier l'absence du support de « canvas » dans un navigateur :

```
<script>
    Modernizr.load({
        test: Modernizr.canvas,
        nope: 'http://flashcanvas.net/bin/flashcanvas.js'
    });
</script>
```

Exemple 2 : Modernizr détecte si le champ de saisie de type « datetime » est supporté par le navigateur. Si ce n'est pas le cas, il affecte aux champs de saisie concernés la fonction « datepicker » du framework jQueryUI (ce qui présuppose que le framework ait été chargé en amont):

```
<script>
    if (!Modernizr.inputtypes.datetime) {
        jQuery("#champ_date1").datepicker({
            dateFormat: 'yy-mm-dd'
        });
        jQuery("#champ_date2").datepicker({
            dateFormat: 'yy-mm-dd'
        });
    }
</script>
```

Exemple 3 : Chargement dynamique de jQuery et jQueryUI, de manière à fournir la possibilité d'afficher un calendrier sur tous les champs de saisie de type « date », à un navigateur qui est dépourvu de cette fonction :

```
<script src="modernizr.js"></script>
<script>
    Modernizr.load({
        test: Modernizr.inputtypes.date,
        nope: [ 'http://ajax.googleapis.com/ajax/libs/jquery/4.4/jquery.min.js',
            'http://ajax.googleapis.com/ajax/libs/jqueryui/8.7/jquery-ui.min.js',
            'jquery-ui.css'],
        complete: function () {
            jQuery('input[type=date]').datepicker({
                dateFormat: 'yy-mm-dd'
            });
        }
    });
</script>
```

Autre atout de Modernizr, il ajoute automatiquement dans le DOM du navigateur un certain nombre de classes CSS, en tenant compte des lacunes du navigateur dans lequel il s'exécute :

- si le navigateur ne supporte pas Javascript, on obtient :

```
<html class="no-js" lang="fr">
```

- si le navigateur supporte Javascript, la liste des classes pourra ressembler à ceci :

```
<html class=" js no-flexbox canvas canvastext webgl no-touch geolocation postmessage
no-websqldatabase indexeddb hashchange history draganddrop websockets rgba hsla
multiplebgs backgroundsize borderimage borderradius boxshadow textshadow opacity
cssanimations csscolumns cssgradients no-cssreflections csstransforms csstransforms3d
csstransitions fontface generatedcontent video audio Localstorage sessionstorage
webworkers applicationcache svg inlinesvg smil svgclippaths">
```

On peut afficher cette liste de classes dans la console de Firebug (via l'onglet « HTML »). Dans l'exemple ci-dessus, on voit – au travers de toutes les classes préfixées par « no- », que le navigateur, en l'occurrence Firefox 21.0, ne supporte pas « flexbox », « touch », « websqldatabase », etc... en revanche il supporte très bien, et nativement, « js » (javascript), « canvas », « webgl », « geolocation », etc...

On peut vérifier via Javascript, que la classe "canvas" est bien présente dans la liste définie par Modernizr :

```
var html=document.querySelector('html');
console.log(html.classList.contains('canvas'));
```

Discussion :

Il n'est pas absolument indispensable de recourir aux « shivs » et à Modernizr.

Les performances des polyfills sont la plupart du temps moins bonnes que les performances de fonctionnalités supportées nativement par le navigateur. De plus, l'utilisation des « polyfills » a des conséquences sur la consommation de bande passante, puisqu'elle augmente la quantité de code javascript transmis au navigateur.

Avant de les utiliser, il convient de les tester avec soin afin de déterminer si leur utilisation est pertinente dans votre contexte.

Dans certains cas, il est peut être plus pertinent d'indiquer à l'utilisateur que son navigateur n'est pas à niveau, et de l'inviter à télécharger une version plus récente, que de tenter de combler les manques à coup de « polyfills ».

2.7 CSS3

La norme HTML5 s'accompagne d'une profonde révision de la norme CSS, avec l'arrivée conjointe de CSS3.

CSS3 apporte beaucoup de nouveautés particulièrement intéressantes pour les webdesigners. Parmi ces nouveautés, il y a en plusieurs qui intéresseront tout autant le développeur Javascript, c'est pourquoi nous allons les étudier dans la suite de ce chapitre. Il s'agit :

- des mediaqueries
- des sélecteurs CSS3
- des effets de dégradés simples à mettre en oeuvre

La norme CSS3 couvre plusieurs autres domaines qui ne seront pas couverts par ce cours, tels que :

- Webfonts
- Affichage "Multi-colonnes"
- Transformations 2D
- Arrières-plans multiples

2.7.1 Mediaqueries et Responsive design

Les "media queries" constituent une des avancées les plus importantes du HTML5, et répondent à un des besoins essentiels des webdesigners, désigné par le terme de "responsive design".

Un "*media query*" est en fait un état logique pouvant prendre 2 positions :

- true : les règles de style définies sur ce media query s'appliquent
- false : elles ne s'appliquent pas

Petit rappel : en CSS 2.1 et HTML 4.01, la syntaxe pour affecter un groupe de directives CSS à un média était la suivante :

```
<link rel="stylesheet" href="foo.css" media="screen">
```

En HTML5, l'attribut "media" prend se voir associer un "media query", ce qui revient schématiquement à écrire ceci :

```
<link rel="stylesheet" href="foo.css" media="screen and (query)">
```

Nous verrons dans un instant ce que l'on peut mettre à la place de (query).

Une autre solution pour déclarer du CSS avec chargement conditionné par "media query"

consiste à le déclarer directement dans le code CSS, de la façon suivante :

```
<style>
    @media screen and (query) { ... }
</style>
```

On notera qu'il est possible d'inclure des feuilles de style externes via la directive @import, de la façon suivante :

```
<style>
@import url('foo.css') screen and (query);
</style>
```

Mais des problèmes de performances ont été relevés avec cette technique, donc il convient de l'utiliser avec prudence. Pour de plus amples informations à ce sujet, on recommandera la lecture de l'article de Steve Souders :

<http://www.smashingmagazine.com/2009/04/09/dont-use-import/>

Chargement du CSS conditionné par la taille de l'écran

Les "media queries" sont généralement utilisés pour charger dynamiquement du code CSS sous condition que les caractéristiques de l'écran répondent aux critères définis dans la requête (query).

Premier exemple :

```
<style>
    @media screen and (width: 480px) {
        body { background-color: white; }
    }
    @media screen and (width: 768px) {
        body { background-color: silver; }
    }
</style>
```

Second exemple :

```
<link rel="stylesheet" href="foo_480.css" media="screen and (width: 480px)">
<link rel="stylesheet" href="foo_768.css" media="screen and (width: 768px)">
```

On peut donc utiliser cette technique pour adapter la présentation de la page aux caractéristiques d'affichage de différents types de terminaux. Par exemple, un texte apparaissant sur plusieurs colonnes dans une page de 768 pixels, pourra être affiché sur une seule colonne dans une page de résolution inférieure, avec éventuellement une autre police, voire une autre taille de caractères.

Vous devez tenir compte de deux ensembles de dimensions: d'abord :

- d'une part, les dimensions du dispositif lui-même,
 - et d'autre part, les dimensions de la zone d'affichage de l'agent sur cet appareil (qui est pour la plupart des utilisateurs un navigateur web, mais la fenêtre peut aussi être une fenêtre d'application). Une personne peut visiter votre site en utilisant une énorme télévision à écran large, mais c'est de peu d'intérêt pour vous si l'application que la personne utilise pour afficher votre site occupe seulement un quart de l'écran. Mais en règle générale, ces deux ensembles de dimensions sont les mêmes sur la plupart des smartphones et tablettes, par exemple, la largeur du navigateur est la même que la largeur de l'appareil.

Cette seconde dimension de la zone d'affichage est communément désignée par le terme de "viewport". Pour préciser le viewport, on indique généralement la dimension horizontale (la largeur en pixels) du viewport, à charge pour le périphérique d'ajuster la résolution horizontale en proportion. C'est la raison pour laquelle on précise rarement - mais il est tout à fait possible de le faire si besoin - le paramètre "height" (hauteur) dans les déclarations telles que celle ci-dessous :

```
<link rel="stylesheet" href="foo.css" media="screen and (width: 480px)">
```

On notera que les dimensions du viewport sont généralement exprimées en pixels, mais qu'il est possible d'utiliser d'autres unités, à condition que les périphériques "cibles" les acceptent.

Les attributs "width" et "height" sont un peu trop restrictifs, on aura tout intérêt à les coupler à, ou à les remplacer par, les attributs "max-width", "min-height", "max-height" et "min-height", comme dans l'exemple suivant :

```
<link rel="stylesheet" href="foo_480.css" media="screen and (max-width: 480px)">
<link rel="stylesheet" href="foo_768.css" media="screen and (min-width: 481px) and
(max-width: 768px)">
<link rel="stylesheet" href="foo_1024.css" media="screen and (min-width: 769px)">
```

On pourra dès lors ajuster, dans chacun des fichiers CSS ci-dessus les dimensions des "div" et autres blocs de données, aux dimensions du viewport considéré.

On peut aussi déclarer des "media queries" directement sur des éléments HTML (ou des classes), l'intérieur même des feuilles de styles :

```
<style>
div1 @media screen and (min-width: 769px) {
  h1 {
    border-style: solid;
    font-size: 2em;
  }
}
div1 @media screen and (max-width: 768px) {
  h1 {
```

```

        border-style: dotted;
        font-size: 3.6em;
    }
}
</style>
```

Nous avons vu les critères de sélection les plus simples, mais il en existe d'autres, éventuellement combinables entre eux :

```

<style>
@media screen and (device-aspect-ratio: 4/3) { ... }
@media screen and (min-aspect-ratio: 8/5) { ... }
@media screen and (orientation: portrait) { ... }
@media all and (orientation: landscape) and (min-width: 800em) { ... }
@media (orientation: landscape) and (min-width: 800em) { ... }
@media (orientation: landscape), (min-width: 800em) {}
@media not all and (device-aspect-ratio: 8/5) {}
</style>
```

Dans le cas des images notamment, il peut être intéressant de prévoir plusieurs résolutions pour chaque image, de manière à pouvoir charger les images dans la résolution la plus adaptée au périphérique considéré :

```

<style>
@media screen and (resolution: 96dpi) {
    E { background-image: url('foo.png'); }
}
@media screen and (min-resolution: 192dpi) {
    E { background-image: url('foo-hires.png'); }
}
@media screen and (min-resolution: 2dppx) {
    E { background-image: url('foo-hires.png'); }
}
</style>
```

On peut aussi adapter le CSS au type de périphérique d'entrée utilisé par l'utilisateur. Par exemple :

```

<style>
@media screen and (pointer: coarse) {
    a { padding: 1em; }
}
</style>
```

Le pointeur "coarse" (en anglais "grossier", au sens "graphique" du terme) utilisé dans l'exemple ci-dessus désigne les périphériques tactiles. Les périphériques plus précis, tels que la souris ou le stilet, sont caractérisés par le mot clé "fine".

Une caractéristique essentielle des périphériques tels que la souris est de permettre le survol (hover) des éléments de la page, ce que ne permettent pas les périphériques tactiles. Media query prend ce problème en considération, avec le mot clé "hover", comme dans l'exemple suivant :

```
<style>
@media screen and (pointer: coarse) {
  a { padding: 1em; }
}
@media screen and (hover:0) {}
</style>
```

Manipulation des Media Queries en JavaScript

Le langage Javascript a été enrichi de méthodes permettant de manipuler les "media queries".

La méthode `matchMedia()` attachée à l'objet "window" permet d'obtenir des informations intéressantes. Testez la dans la console de votre navigateur :

```
window.matchMedia('screen and (min-width: 800px)');
/*
Le résultat obtenu est un objet de type MediaQueryList :
- Résultat obtenu avec une fenêtre d'affichage de taille supérieure ou égale à 800
pixels :
  MediaQueryList {matches: true, media: "screen and (min-width: 800px)", 
addListener: function, removeListener: function}
- Résultat obtenu avec une fenêtre d'affichage de taille supérieure ou égale à 800
pixels :
  MediaQueryList {matches: false, media: "screen and (min-width: 800px)", 
addListener: function, removeListener: function}
*/
```

On peut dès lors exploiter l'information pour tout un tas d'usages :

```
var mq = window.matchMedia('screen and (min-width: 800px)');
if (mq.matches) {
  // faire quelque chose
} else {
  // faire autre chose
}
```

Recommandations :

Les "media queries" rencontrent un grand succès dans le monde du webdesign, pour leur

souplesse, mais aussi parce qu'ils sont reconnus par la grande majorité des terminaux équipés de navigateurs récents. Si vous souhaitez les utiliser pour des applications "métier", il convient de faire des tests poussés, pour s'assurer que les terminaux cibles de vos applications les supportent.

Une des recommandations faites par certains gourous du webdesign, qu'il convient de prendre en considération, est la suivante : il est préférable de concevoir l'interface utilisateur en premier lieu pour les périphériques d'affichage cibles ayant le plus petit viewport. Une fois que l'interface utilisateur est au point, on peut compléter le design en ajoutant des caractéristiques propres aux périphériques offrant des dimensions supérieures. Il semblerait que dans la pratique, la méthode inverse - consistant à concevoir l'application pour le viewport le plus élevé, puis à la "dépouiller" au fur et à mesure que l'on diminue le viewport - donne des résultats moins satisfaisants.

2.7.2 Sélecteurs CSS3

Les sélecteurs CSS3 constituent une autre avancée majeure de la norme HTML5, car ils intègrent la technique dite des "pseudo-classes", technique qui permet d'effectuer des sélections très fines au niveau des éléments de la page HTML. Ces pseudo-classes peuvent être utilisées à la fois dans les feuilles de styles (CSS), mais aussi en Javascript (via les méthodes querySelector() et querySelectorAll() que nous étudierons plus loin).

Nous allons étudier dans ce chapitre les principales pseudo-classes disponibles.

La pseudo-class :nth-child()

La notation :nth-child(an+b) désigne un élément de la page qui a "an+b-1" frères et soeurs (en anglais : "siblings") devant lui dans l'arborescence du DOM. Plutôt que de s'apresantir sur une définition inutilement compliquée, étudions quelques exemples :

```
tr:nth-child(2n+1) /* sélectionne toutes les lignes impaires d'un tableau HTML */  
tr:nth-child(odd) /* idem */  
tr:nth-child(2n+0) /* sélectionne toutes les lignes paires d'un tableau HTML */  
tr:nth-child(even) /* idem */  
  
<style>  
/* Exemple : sur un groupe de 4 paragraphes, appliquer des couleurs différentes sur  
chacun des paragraphes */  
p:nth-child(4n+1) { color: navy; }  
p:nth-child(4n+2) { color: green; }  
p:nth-child(4n+3) { color: maroon; }  
p:nth-child(4n+4) { color: purple; }
```

```
</style>
```

Autres exemples avec le signe moins :

```
:nth-child(10n-1) /* sélectionne les 9ème, 19ème, 29ème, etc, éléments */
:nth-child(10n+9) /* idem */
:nth-child(10n+-1) /* syntaxe invalide, sera ignorée par le navigateur */
```

Cas particulier avec :nth-child(*b*).

Exemples:

```
foo:nth-child(0n+5) /* sélectionne un élément foo qui est 5ème enfant du parent
auquel il appartient */
foo:nth-child(5) /* idem */
```

Si *a*=1, ou *a*=-1, alors il peut être omis.

```
bar:nth-child(1n+0) /* sélectionne tous les éléments bar */
bar:nth-child(n+0) /* idem */
bar:nth-child(n) /* idem */
bar /* idem */
```

Attention à la gestion des espaces dans les critères de sélection :

```
/* Examples valides: */
:nth-child( 3n + 1 )
:nth-child( +3n - 2 )
:nth-child( -n+ 6)
:nth-child( +6 )

/* Examples invalides: */
:nth-child(3 n)
:nth-child(+ 2n)
:nth-child(+ 2)
```

Autre exemple :

```
tr:nth-child(-n+6) /* sélectionne la 6ème ligne d'un tableau HTML */
```

La pseudo-classe :nth-last-child() :

La notation :nth-last-child(*an+b*) désigne un élément de la page qui a "*an+b-1*" frères (siblings) **derrière lui** dans l'arborescence du DOM.

Exemples :

```
tr:nth-last-child(-n+2) /* sélectionne les 2 dernières lignes d'un tableau HTML */
li:nth-last-child(odd) /* sélectionne toutes les lignes impaires d'une liste "li"
en partant de la dernière */
```

La pseudo-classe :nth-of-type() :

La notation de pseudo-classe `:nth-of-type(an+b)` sélectionne un élément qui a $an+b-1$ éléments "frères et soeurs" (siblings) **de même type** devant lui dans l'arborescence du DOM.

Exemple :

```
/* Alternance d'image paires et impaires, à droite et à gauche de la page */
img:nth-of-type(2n+1) { float: right; }
img:nth-of-type(2n) { float: left; }
```

La pseudo-classe :nth-last-of-type()

La notation `:nth-last-of-type(an+b)` représente un élément qui a $an+b-1$ frères et soeurs de même type **après** lui dans l'arborescence du DOM.

Exemple : Pour représenter tous les enfants de type "h2" dans l'arborescence de l'élément "body", excepté le premier et le dernier, on peut utiliser la syntaxe suivante

```
body > h2:nth-of-type(n+2):nth-last-of-type(n+2)
body > h2:not(:first-of-type):not(:last-of-type) /* idem */
```

La pseudo-classe :first-child

Equivaut à `:nth-child(1)`.

La pseudo-classe `:first-child` représente un élément qui est le premier enfant d'un groupe sélectionné.

Par exemple, le sélecteur suivant sélectionne un élément "p" qui est le premier enfant d'un élément "div".

```
div > p:first-child
```

Sur le bloc de code HTML ci-dessous, le sélecteur ci-dessus sélectionnera l'élément "p" qui se trouve à l'intérieur de la "div" :

```
<p> The last P before the note.</p>
<div class="note">
  <p> The first P inside the note.</p>
</div>
```

Mais ce même sélecteur ne sélectionnera pas l'élément "p" à l'intérieur de l'élément "div" dans l'exemple de code HTML ci-dessous :

```
<p> The last P before the note.</p>
<div class="note">
  <h2> Note </h2>
  <p> The first P inside the note.</p>
</div>
```

La pseudo-classe :last-child

Equivaut à :nth-last-child(1).

La pseudo-classe :last-child représente un élément qui est le dernier enfant d'un élément sélectionné précédemment.

Par exemple, le sélecteur suivant sélectionne le dernier élément "li" d'une liste non ordonnée ("ol").

```
ol > li:last-child
```

La pseudo-classe :first-of-type

Equivaut à :nth-of-type(1).

La pseudo-classe :first-of-type représente un élément qui est le premier frère (ou soeur) de son type, dans la liste des enfants d'un élément parent sélectionné.

Par exemple, le sélecteur suivant représente le premier élément "dt" d'une liste de définition "dl", cet élément "dt" étant le premier élément de son type dans la liste des enfants de l'élément "dl".

```
dl dt:first-of-type
```

Ce sélecteur permettra de récupérer les éléments "dt gigogne" et "dt fusée", mais pas l'élément "dt table".

```
<dl>
  <dt>gigogne</dt>
  <dd>
    <dl>
      <dt>fusée</dt>
      <dd>multistage rocket</dd>
      <dt>table</dt>
      <dd>nest of tables</dd>
    </dl>
  </dd>
</dl>
```

La pseudo-classe :last-of-type

Equivaut à :nth-last-of-type(1).

La pseudo-classe :last-of-type représente un élément qui est le dernier frère (ou soeur) de son

type, dans la liste des enfants d'un élément parent sélectionné.

Par exemple, le sélecteur suivant sélectionne la dernière cellule "td" de la ligne "tr" :

```
tr > td:last-of-type
```

A noter : on retrouvera ces notions de sélecteurs et de pseudo-classes, de manière plus détaillée, dans le chapitre dédié au DOM, ainsi que dans l'annexe (avec un tableau récapitulatif complet, ainsi qu'un chapitre d'exercices).

2.7.3 Effets de dégradés

La norme CSS3 apporte la possibilité de pouvoir générer des effets de dégradés à la place des banales couleurs de remplissage disponibles en CSS2. Le présent chapitre est une introduction à l'utilisation de ces effets.

Avec CSS3, on peut désormais obtenir des effets de dégradés linaires :

```
background-image: linear-gradient(to top left, #C24704, #FFEB79);
```

On peut aussi obtenir des effets de dégradés radiaux :

```
background-image: radial-gradient(circle farthest-corner at 100px 50px, #C24704, #FFEB79 35%, #00ADA7);
```

Ou encore des effets de motifs assez sophistiqués :

```
background-image: repeating-linear-gradient(-45deg, #426A77, #FFF 6px);
```

On peut également définir des effets d'arrondi, exprimés en pourcentages ou en pixels :

```
border-radius: 20%;  
border-radius: 3px;
```

Les éditeurs de certains navigateurs avaient fait le pari de commencer à intégrer ces nouveaux filtres, avant que la spécification CSS3 ne soit complètement finalisée. N'étant pas certains de l'implémentation finale de certains effets, ils décidèrent d'adopter une convention consistant à préfixer les nouveaux filtres CSS comme dans l'exemple suivant :

```
background-image: -moz-linear-gradient(315deg, #C24704, #FFEB79); /* Mozilla (Firefox) */  
background-image: -o-linear-gradient(315deg, #C24704, #FFEB79); /* Opera */  
background-image: -webkit-gradient(linear, 0% 0%, 0% 100%, from(#C24704), to(#FFEB79)); /* Webkit (Chrome et Safari) */  
background-image: linear-gradient(135deg, #C24704, #FFEB79); /* Spécification CSS3 standard */
```

Sur Firefox v27 et Google Chrome v32, les préfixes indiqués ci-dessus ne sont plus nécessaires. En revanche, sur IE9, ces effets sont inopérants. Ils devraient fonctionner avec IE10, mais cela reste à vérifier.

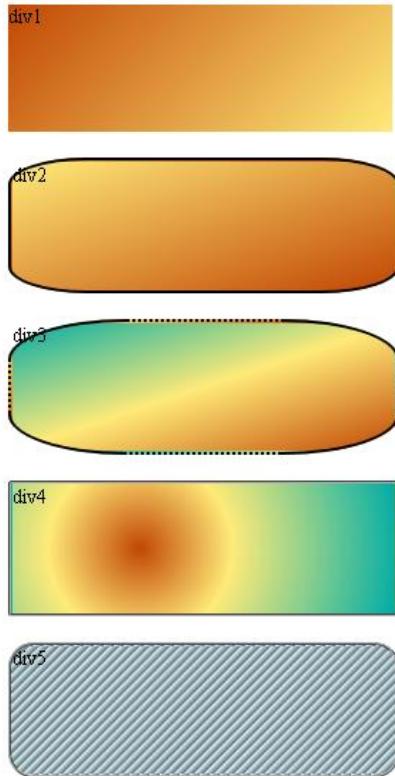
Exemple d'utilisation de ces effets sur plusieurs div :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<head>
<title>Insert title here</title>
</head>
<style>
div {
    width: 300px;
    height: 100px;
    margin: 20px;
}
.div1 {
    background-image: linear-gradient(#C24704, #FFEB79);
    border-radius: 10%;
    border-style: dashed;
}
.div2 {
    background-image: linear-gradient(to top left, #C24704, #FFEB79);
    border-radius: 20%;
    border-style: solid;
}
.div3 {
    background-image: linear-gradient(to top left, #C24704, #FFEB79, #00ADA7);
    border-radius: 30%;
    border-style: dotted;
}
.div4 {
    background-image: radial-gradient(circle farthest-corner at 100px 50px,
#C24704, #FFEB79 35%, #00ADA7);
    border-radius: 3px;
    border-style: groove;
}
.div5 {
    background-image: repeating-linear-gradient(-45deg, #426A77, #FFF 6px);
    border-radius: 20px;
    border-style: ridge;
}

</style>
<body>
<div class="div1">div1</div>
<div class="div2">div2</div>
<div class="div3">div3</div>
```

```
<div class="div4">div4</div>
<div class="div5">div5</div>
</style>
</body>
</html>
```

Les résultats obtenus dans Firefox et Google Chrome sont assez similaires :



On le voit, ces nouveaux effets permettent à des développeurs webs peu à l'aise avec le webdesign, d'obtenir des effets esthétiques, avec seulement quelques lignes de code CSS.

Pour approfondir le sujet, les développeurs gagneront à étudier le code CSS du Bootstrap de Twitter, qui est riche en effets de ce genre.

Avant de décider de les utiliser de manière intensive, il conviendra de vérifier la compatibilité de ces effets avec les navigateurs cibles.

3 Bases du langage Javascript



ECMAScript (plus connu sous le nom de Javascript) est un langage de programmation de type "langage de script", standardisé par Ecma International (ECMA pour "European Computer Manufacturers Association") dans le cadre de la spécification ECMA-262. Il s'agit donc d'un standard, dont les spécifications sont mises en œuvre dans différents langages de script, comme JavaScript ou ActionScript, ainsi qu'en C++ (norme 2011). C'est un langage de programmation orienté prototype.

Pour tout savoir sur l'histoire de ECMAScript :

<http://fr.wikipedia.org/wiki/ECMAScript>

Javascript est sans conteste le langage du web, mais il est aussi en passe de devenir le langage de développement phare pour le développement d'applications sous de nombreux systèmes d'exploitation, tels que : Windows 8, Google Androïd, iOS, GNOME Shell (Linux Debian, Fedora, etc...). Grâce à la montée en puissance rapide de NodeJS, le langage Javascript est aussi devenu un langage pertinent pour le développement d'application côté serveur.

Javascript est un langage inhabituel dans son approche de la programmation objet, ainsi que dans son modèle évènementiel. Ces paris technologiques audacieux se sont finalement révélés gagnants, car ils confèrent à Javascript une souplesse et une puissance impressionnante, comme on le découvrira tout au long de ce cours.

Avertissement : jusqu'à sa version 1.4, ce support était focalisé essentiellement sur la norme ECMAScript 5 (ES5), avec quelques éléments relatifs à ES6 disséminés ici et là. A partir de la version 1.5 de ce même support, la présentation des spécificités de ES6 est plus systématique (mais beaucoup reste à faire avant que ce support ne couvre ES6 de manière exhaustive).

Il faut souligner que la norme ECMAScript 6 (ES6), apparue en 2015, est souvent désignée par le terme ES2015 dans les documentations disponibles sur internet. ES6 est donc équivalent à ES2015, ce qui peut entraîner une confusion dans l'esprit des développeurs.

Il faut également souligner que ES6 est partiellement supporté par les navigateurs les plus récents, mais ce support n'est pas homogène, ce qui peut poser quelques soucis sur des applications en production. Nous reviendrons dans un chapitre ultérieur sur ces problématiques.

3.1 Commentaires

La saisie de commentaires à l'intérieur du code source obéit aux mêmes règles que celles pratiquées dans les langages PHP et Java.

La saisie de commentaire sur une seule ligne se fait en règle générale en utilisant les 2 slashes :

```
//Commentaire sur une seule ligne
```

Mais il est aussi tout à fait correct d'utiliser la syntaxe suivante, aussi bien pour une seule ligne que pour plusieurs :

```
/* Exemple 1 : Commentaire sur une seule ligne */

/* Exemple 2 : Commentaire sur
deux lignes */

/*
Exemple 3 : Commentaire sur deux lignes également
(généralement préféré à l'exemple 2, car moins fouilli)
*/

/*
* Exemple 4 : Style de commentaire multi-lignes couramment pratiqué
* en PHP et en Java, que l'on préférera aux exemples 2 et 3
*/
```

Les développeurs Javascript peuvent générer la documentation de leurs applications en utilisant des solutions qui s'inspirent toutes deux de l'application Javadoc utilisée par les développeurs Java (qui a aussi été déclinée sous plusieurs formes en PHP). Ces solutions sont les suivantes :

- le Toolkit JSDoc (<http://code.google.com/p/jsdoc-toolkit/>)
- YUIDoc de Yahoo (<http://yuilibrary.com/projects/yuidoc>).

On peut enrichir la documentation d'un certain nombre de tags qui seront repris par les générateurs de documentation. Ces tags sont les suivants :

@namespace

L'environnement de référence contenant l'objet

@class

Nom de l'objet ou du constructeur ayant servi à instancier l'objet courant

@method

Définit une méthode à l'intérieur d'un objet

@param

Liste des paramètres de la fonction. Le type des paramètres est indiqué entre accolades, suivi

par le nom du paramètre et sa description
@return
Définit la valeur renvoyée par la méthode

Exemple d'implémentation :

```
/**  
 * Construction de l'objet Person  
 * @class Person  
 * @constructor  
 * @namespace MYAPP  
 * @param {String} first Nom  
 * @param {String} last Prénom  
 */  
MYAPP.Person = function(first, last) {  
    /**  
     * Nom de la personne  
     * @property first_name  
     * @type String  
     */  
    this.first_name = first;  
    /**  
     * Prénom de la personne  
     * @property last_name  
     * @type String  
     */  
    this.last_name = last;  
};  
/**  
 * Méthode getName attachée au constructeur et renvoyant le nom et le prénom de la  
personne  
*  
* @method getName  
* @return {String} Le nom de la personne  
*/  
MYAPP.Person.prototype.getName = function() {  
    return this.first_name + ' ' + this.last_name;  
};
```

3.2 Types de variables "primitifs"

3.2.1 Introduction

Certains aspects de Javascript peuvent être déroutants pour les développeurs habitués à des langages fortement typés, comme Java par exemple.

Javascript partage avec PHP la particularité d'offrir des variables faiblement typées, et dont le type peut très facilement changer durant le cycle de "vie" d'un script.

Javascript propose 3 type de données primaires (les anglophones emploient le terme de "literal", ou encore de "primitive"), qui sont : string, numeric, et boolean.

Mais Javascript met également à disposition 3 objet prédéfinis qui sont : String, Number et Boolean.

Cet aspect est souvent mal compris des développeurs débutant en Javascript.

Pour bien l'expliquer, prenons un premier exemple :

```
var prenom = "Gregory";
var prenomMajuscule = prenom.toUpperCase();
```

Dans l'exemple ci-dessus, on a défini une variable primitive de type "string", la variable "prenom". La variable prenom n'étant pas un objet, on pourrait penser que l'on n'a pas le droit d'écrire ceci : prenom.toUpperCase() .

Mais en réalité, Javascript nous autorise à le faire, car il va automatiquement encapsuler la variable "prenom" dans un objet "wrapper", l'objet "String". La variable "prenom" change donc instantanément de type, elle devient un objet "String", ce qui lui permet d'utiliser la méthode de conversion en majuscule "toUpperCase()", méthode standard de l'objet "String". Dès que la méthode "toUpperCase()" a fini son travail, l'objet disparaît et la variable "prenom" retrouve instantanément son type primitif initial, qui était en l'occurrence le type "string".

La déclaration de variables se fait au moyen de l'instruction "var".

Quelques exemples :

```
function doSomething() {
    var result = 10 + value;
    var value = 10;
    return result;
}
```

Dans l'exemple ci-dessus, la variable "value" est utilisée avant d'être initialisée, mais la variable "result" contiendra à l'arrivée la valeur NaN (Not a Number). Pour bien comprendre, voici comment Javascript traduit dans les coulisses le code ci-dessus :

```
function doSomething() {
    var result;
    var value;
    result = 10 + value;
    value = 10;
    return result;
}
```

Explication : la déclaration des 2 variables "result" et "value" est automatiquement "hissée" au début de la fonction (les anglophones emploient le terme "hoisted"). C'est seulement quand les variables sont déclarées, que la fonction procède à l'exécution des opérations d'affectation (comme 10+value). Ce genre de subtilité, si elle est mal comprise, peut faire perdre beaucoup de temps en débogage.

Pour assurer une bonne maintenabilité du code, il est préférable de déclarer toutes les variables au début de la fonction, si possible en attribuant à chacune une valeur par défaut. Le mécanisme de "hoisting" de Javascript fait la même chose, mais le fait de disséminer la déclaration des variables dans le code peut être à l'origine d'erreurs difficiles à détecter.

3.2.2 Strings

3.2.2.1 Introduction

La déclaration des chaînes de caractères est similaire à ce que l'on connaît dans d'autres langages.

```
var strString = "Ceci est une chaîne";
var anotherString= 'Au fait, ça aussi';
```

Contrairement à PHP qui offre un mécanisme de parsing et de substitution de variables sur les chaînes déclarées avec des guillemets, Javascript ne fait pas de différence entre une chaîne définie avec des guillemets ou des apostrophes.

```
var string_value = "Exemple de 'chaîne' avec des apostrophes à l'intérieur." ;
var string_value = 'Exemple de "chaîne" avec des guillemets à l\'intérieur.' ;
```

Le mécanisme d'échappement de certains caractères est similaire à celui que l'on trouve dans d'autres langages comme PHP.

Caractères d'échappement :

\' Simple quote

```
\" Double quote
\\ Antislash
\b Caractère de retour
\f Form feed
\n Nouvelle ligne
\r Retour chariot
\t Tabulation horizontale
```

On l'a évoqué dans l'introduction, Javascript offre un mécanisme de type "wrapper" transformant une chaîne "littérale" (ou "primitive" dans le jargon des anglophones), en un objet de type String. Ce mécanisme est déclenché automatiquement par Javascript dès que l'on fait appel, sur une variable "primitive", à une méthode de l'objet String, comme par exemple les méthodes `toUpperCase()` et `toLowerCase()`.

Attention : les deux déclarations ci-dessous sont strictement équivalentes, tant que l'on n'utilise pas le constructeur "new" devant la fonction `String()` :

```
var musicien = "Sidney Bechet";
console.log(musicien); // Sidney Bechet
var musicien = String("Sidney Bechet");
console.log(musicien); // Sidney Bechet
```

Le mécanisme de conversion de chaîne en objet (et inversement) étant coûteux en ressources, si l'on fait beaucoup appel à des méthodes de l'objet String, on peut trouver plus judicieux de créer dès le départ un objet String. Dans ce cas très particulier, on aura recours au constructeur "new" :

```
var musicien = new String("Sidney Bechet");
console.log(musicien); // String { 0="S", 1="i", 2="d", more...}
console.log(musicien.toLowerCase()); // sidney bechet
```

La fonction `String()` peut aussi être utilisée pour convertir une valeur numérique en type "string" :

```
var valeur1 = String(60);
var valeur2 = +valeur1 ;
console.log(valeur2) ; // 60
console.log(typeof valeur1) ; // string
console.log(typeof valeur2) ; // number
var valeur3 = valeur2 + ' ' ;
console.log(valeur3) ; // 60
console.log(typeof valeur3) ; // string
```

Explications :

Le fait d'utiliser la fonction `String()` sur une valeur numérique a pour effet de la convertir en type "string". Mais on peut aussi obtenir le même effet en effectuant une simple concaténation

d'une valeur numérique avec une chaîne de caractères vide (comme dans le calcul de la variable "valeur3"). Dans le cas du calcul de la variable "valeur1", on parlera de conversion explicite, alors que dans le cas du calcul de "valeur3", on parlera de conversion implicite.

De même, le fait d'utiliser la fonction Number() sur une chaîne de caractères a pour effet d'en fait une variable de type Number. Mais l'utilisation d'un simple "+" devant une chaîne de caractères a strictement le même effet que la fonction Number().

Point très important :

Le fait de créer une variable de type String (sans utiliser le constructeur new) n'empêche nullement de recourir aux méthodes associées à l'objet String. L'exemple ci-dessous le démontre clairement :

```
var musicien = String("Sidney Bechet");
console.log(musicien); // Sidney Bechet
console.log(musicien.toLowerCase()); // sidney bechet
```

Ce qui se passe ici est relativement simple à comprendre : dès lors que l'on fait appel à une méthode de l'objet String, comme ici la méthode toLowerCase(), l'interpréteur Javascript effectue un transtypage dynamique de la variable "musicien", du type String vers un "objet String". Il utilise la méthode invoquée (ici la méthode toLowerCase()) puis redonne à la variable "musicien" son type d'origine (en l'occurrence le type String). Ce mécanisme est très rapide, et transparent pour le développeur JS.

3.2.2.2 Concaténation de chaînes

La concaténation de chaînes de caractères se fait très simplement au moyen de l'opérateur "+" :

```
var string1 = "Ceci est un ";
var string2 = "test";
var string3 = string1 + string2;
console.log(string3); // Ceci est un test
string3 += " raté";
console.log(string3); // Ceci est un test raté
```

On peut également utiliser la méthode concat(), appliquée à une chaîne, éventuellement créée à la volée, comme dans l'exemple suivant :

```
var nwStrng = "".concat("Ceci ","est ","une ","chaîne");
console.log(nwStrng); // Ceci est une chaîne
```

Avec des types différents, cela fonctionne également, mais attention à l'ordre des variables concaténées, car cela peut avoir certains effets de bord :

```
var valtotal = 223.45;
var strtotal = "Total = " + valtotal;
console.log(strtotal); // Total = 223.45
var strValue1 = "4" + 2 + 1;
console.log(strValue1); // "421"
var strValue2 = 4 + 2 + "1";
console.log(strValue2); // 61
```

On peut aussi utiliser la méthode `join()` sur un tableau pour concaténer ses éléments sous forme d'une chaîne :

```
var fruitArray = ['pomme', 'poire', 'abricot', 'citron'];
var resultString = fruitArray.join('-');
console.log(resultString); // pomme-poire-abricot-citron
```

3.2.2.3 Manipulation de sous-chaîne

L'extraction d'une portion de chaîne à l'intérieur d'une chaîne plus grande se fait très simplement au moyen de la méthode `substring()`. Couplée avec la méthode `indexOf()` pour détecter la position de certains caractères, on peut obtenir des résultats intéressants :

```
var sentence = "Voici une liste: kiwis, oranges, pommes, bananes.";
var start = sentence.indexOf(":");
console.log(start); // 15
var end = sentence.indexOf(".", start+1);
console.log(end); // 48
var list = sentence.substring(start+1, end);
console.log(list); // kiwis, oranges, pommes, bananes
```

La méthode `indexOf()` offre un second paramètre permettant d'indiquer la position de départ de la recherche :

```
var tstString = "Cette voiture est ma voiture";
var iValue = tstString.indexOf("voiture", 10);
console.log(iValue); // 21
```

La méthode `lastIndexOf()` permet de trouver la position de la dernière occurrence d'une chaîne :

```
var iValue2 = tstString.lastIndexOf("voiture");
console.log(iValue2); // 21
```

La méthode `split()` est très pratique pour transformer une chaîne en tableau, sur la base d'un pattern de séparation :

```
var fruits = list.split(", ");
console.log(fruits); // ["kiwis", "oranges", "pommes", "bananes"]
```

3.2.2.4 Suppression des espaces (trim)

La norme ECMAScript5 apporte une nouvelle méthode trim() permettant de supprimer les blancs situés en début et en fin de chaîne :

Exemple d'utilisation :

```
<div id="test">chrysler, volvo, toyota , renault ,citroën, peugeot, mercedes</div>

<script>
    var resultString = "";
    var i, i_len;
    var strng = '';
    for (i = 0, i_len = lines.length ; i < i_len; i++) {
        strng = lines[i].trim();
        resultString += strng + "-";
    }
    console.log(resultString); // chrysler-volvo-toyota-renault-citroën-peugeot-
mercedes-
</script>
```

Avant ECMAScript 5, il fallait recourir à une expression régulière pour obtenir le même effet que la méthode trim().

Beaucoup de navigateurs reconnaissent aujourd'hui la méthode trim(), mais certains navigateurs plus anciens comme IE8 ne la reconnaissent pas.

Comment s'assurer que le navigateur supporte la méthode trim() ? Et comment ajouter à l'objet String une méthode trim() manquante ?

```
if (typeof String.trim == "undefined") {
    String.prototype.trim = function() {
        return this.replace(/(^s*)|(\s*$)/g, "");
    }
}
```

On notera que la norme ECMAScript 5 fournit également les méthodes trimLeft() et trimRight().

3.2.2.5 Fonctions sur les chaînes

Tableau récapitulatif des méthodes applicables à l'objet String :

Méthode	Description	Paramètres
valueOf	Renvoie le littéral correspond à l'objet String	Néant
length	Propriété (et non pas méthode) renvoyant la longueur du littéral	à utiliser sans parenthèses
anchor	Crée une ancre HTML	String with anchor title
big, blink, bold, italics, small, strike, sub, sup	Renvoie la balise HTML	Néant (à utiliser avec parcimonie, certaines balises étant dépréciées en HTML5)
charAt, charCodeAt	Renvoie soit un caractère (charAt) soit le code de caractère (charCodeAt) à une position donnée	Integer representing position, starting at position zero (0)
indexOf	Renvoie la position de départ de la première occurrence de la sous-chaîne recherchée	Chaîne de recherche
lastIndexOf	Renvoie la position de départ de la dernière occurrence de la sous-chaîne recherchée	Chaîne de recherche
link	Renvoie HTML pour le lien	URL pour l'attribut href
concat	concatène plusieurs chaînes ensemble	chaînes à concaténer, séparées par une virgule
split	Transforme la chaîne en tableau, sur la base d'un séparateur	le séparateur, suivi du nombre maximum d'éléments à envoyer dans le tableau résultant
slice	Renvoie une portion de chaîne	Début et fin de position de la chaîne recherchée
substring, substr	Renvoie une sous-chaîne	Début et fin de position de la chaîne recherchée
match, replace, search	Expressions régulières : match, replace, and search	String with regular expression
toLowerCase, toUpperCase	Conversion en minuscule et majuscule	Néant
trim, trimLeft, trimRight	Suppression des blancs	Néant

3.2.3 Boolean

Le type de données "booléen" accepte 2 valeurs : true et false.

Ces valeurs ne doivent pas être encapsulées dans des guillemets ou des apostrophes (sinon ce sont des chaînes et non plus des booléens).

```
var estActif = true;
var hasChildren = false;
```

On peut utiliser la fonction Boolean() pour évaluer la valeur d'une variable :

```
var someValue = 0;
var someBool = Boolean(someValue);
console.log(someBool) ; // false
```

Une autre approche pour l'évaluation d'un booléen, en utilisant la double négation :

```
var strValue = "1";
var numValue = 0;
var boolValue = !!strValue; // convertit "1" à true
boolValue = !!numValue; // convertit 0 à false
```

Et bien évidemment, on peut écrire des tests du genre...

```
if (boolValue) {
    ...
}
```

3.2.4 Number et Math

3.2.4.1 Number

Certains auteurs d'ouvrages dédiés à Javascript prétendent que les nombres sont de type "entier" ("integer") s'ils ne contiennent pas de décimale, ou de type "virgule flottante" ("float") dans le cas contraire. Mais il s'agit d'un mauvais raisonnement, qui traduit une méconnaissance du fonctionnement de Javascript. En effet, le développeur JS n'a accès qu'à un seul type de valeur numérique, le type "Number", peu importe que le nombre qu'il crée contienne une partie décimale ou pas.

Exemple de variable de type "Number" :

```
var someValue = 10; // treated as integer 10, in base 10
var myNum = 3.18;
var newNum = myNum * someValue;
```

On peut créer des nombres en utilisant l'objet Number via la méthode "constructeur" (new) :

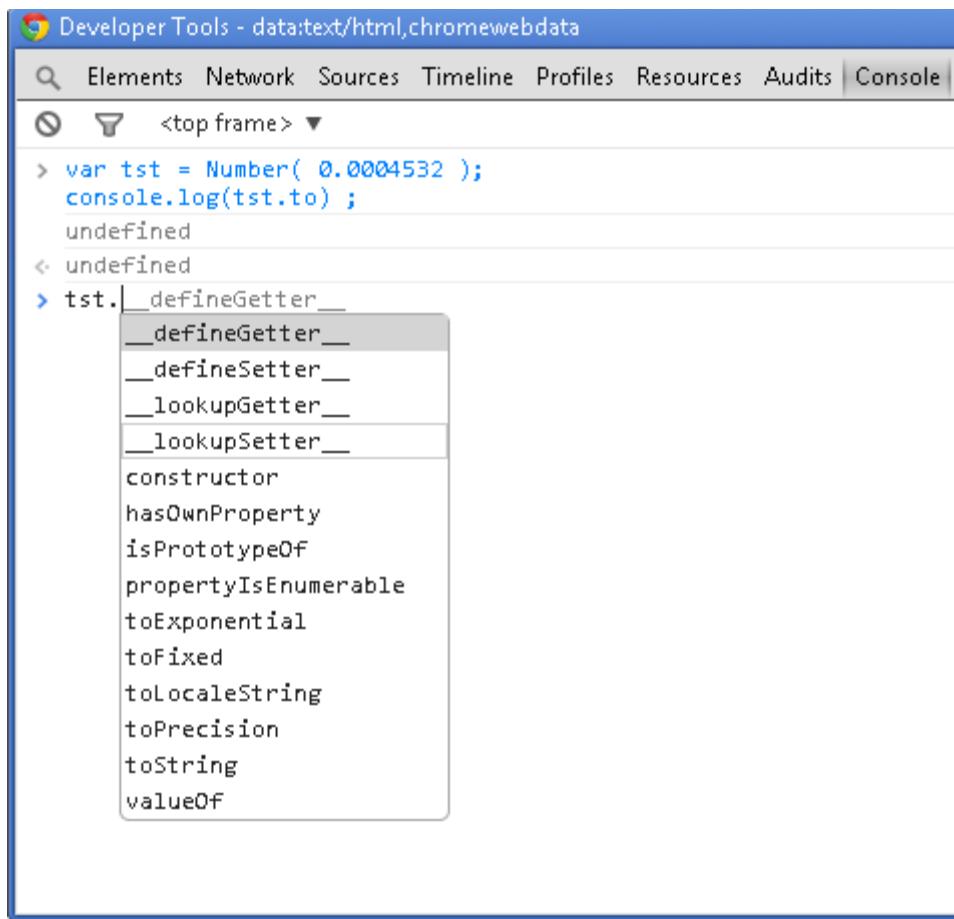
```
var newNum = new Number(23);
```

Mais sauf cas très particulier, on recommandera d'utiliser Number() plutôt comme une fonction que comme un objet, car l'encapsulation d'une valeur numérique sous forme d'objet est plus coûteuse en ressources. Il sera donc préférable d'utiliser autant que possible la syntaxe suivante :

```
var newNum = Number(23); // 23
var newNum = Number('23'); // 23
```

Dans l'exemple ci-dessus, les 2 lignes renvoient strictement la même valeur.

L'auto-complétion proposée par la console de Google Chrome permet de prendre connaissance des méthodes disponibles avec l'objet Number. On remarque d'ailleurs que, même si la variable "tst" a été créée via la fonction Number() sans constructeur, les méthodes de l'objet Number() sont néanmoins omniprésentes :



```
var tst = Number( 0.0004532 );
console.log(tst.toFixed()); // 0
console.log(tst.toPrecision(9)); // 0.00045320000
console.log(tst.toExponential()); // 4.532e-4
console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
console.log(Number.MIN_VALUE); // 5e-324
```

Dans l'exemple ci-dessus, on a créé une variable de type "Number" que l'on a appelée "tst", et on constate que l'on dispose des différentes méthodes associées à l'objet Number, alors même que l'on n'a pas créé d'objet de type Number (puisque l'on n'a pas utilisé le constructeur "new"). En fait, l'interpréteur effectue un transtypage dynamique de la variable `tst`, du type "Number" vers un "objet Number". Ce transtypage est effectué à la volée, le temps d'utiliser la méthode souhaitée (par exemple `toFixed()`), puis l'objet reprend instantanément son type d'origine (qui était le type "Number"). Ce transtypage dynamique est très rapide, et en règle générale, on n'a pas besoin de se préoccuper des problèmes de performances que celui induit.

Les fonctions `parseInt()` et `parseFloat()` sont très pratiques pour convertir des chaînes en nombres :

```
console.log(parseInt("3.5")); // 3
console.log(parseFloat("3.5")); // 3.5
console.log(parseFloat("3.5a")); // 3.5
console.log(parseFloat("a3.5")); // NaN
console.log(parseInt("trois")); // NaN
```

On voit que dans certains cas, la conversion vers le type "Number" peut échouer et renvoyer la valeur `Nan`. Nous en reparlerons un peu plus loin dans ce chapitre.

Pour rappel, on avait vu dans le chapitre d'introduction sur les "Strings" qu'il existait plusieurs méthodes pour convertir une chaîne en nombre (avec la fonction `Number()` ou l'opérateur `+`) :

```
var valeur1 = String(60);
var valeur2 = +valeur1 ;
console.log(valeur2) ; // 60
console.log(typeof valeur1) ; // string
console.log(typeof valeur2) ; // number
var valeur3 = valeur2 + ' ' ;
console.log(valeur3) ; // 60
console.log(typeof valeur3) ; // string
var valeur4 = Number(valeur3) ;
console.log(valeur4) ; // 60
console.log(typeof valeur4) ; // number
```

On pourra utiliser cette fonction de conversion pour récupérer les valeurs d'un tableau HTML en numérique :

```
var rows = document.getElementById("table1").children[0].rows;
var numArray = [];
var i, i_len ;
for (i = 0, i_len = rows.length; i < i_len; i++) {
    numArray[i] = parseInt(rows[i].cells[1].firstChild.data);
}
```

Attention : l'expression `NaN` renvoyée par l'interpréteur JS suite à un calcul incorrect (ou à un résultat erroné renvoyé par les fonctions `parseInt()` ou `parseFloat()`) est souvent traduit par l'expression "Not a Number" (n'est pas un nombre). Mais il est préférable de le traduire par "Mauvais nombre" ou "Nombre invalide", car du point de vue de Javascript, il s'agit bien d'un nombre, comme le démontre le test suivant :

```
var a = 2 / "foo"; // NaN
typeof a === "number"; // true
```

Cette mauvaise compréhension du terme `NaN`, de la part des développeurs JS, entraîne parfois des erreurs de codage.

Pour ajouter à la confusion, il existe une fonction `isNaN()`, qui est associée à l'objet Window. On serait donc tenté d'écrire ceci :

```
var a = 2 / "foo";
var b = "foo";
console.log(a); // NaN
console.log(b); "foo"
console.log(window.isnan( a )); // true
console.log(window.isnan( b )); // true -- Oups, ce résultat est absurde !
```

On voit avec le test sur la variable "b" que la fonction `isNaN` ne fait pas ce à quoi l'on s'attendait, en fait elle vérifie simplement si les variables qu'on lui transmet sont de type "Number"... ou pas. En fait cette fonction est de peu d'intérêt et il est préférable de la bannir de notre arsenal de développeur JS.

Ce problème a été relevé par Kyle Simpson dans son excellent livre :

"You don't know JS : Types and Grammars", édité chez O'Reilly.

Dans son livre, Kyle Simpson recommande d'utiliser plutôt la fonction `isNaN` qui sera associée à l'objet Number à partir de la version 6 de la norme ECMAScript. Certains navigateurs implémentent déjà cette fonction, mais si ce n'est pas le cas du vôtre, vous pouvez recourir au polyfill suivant pour l'implémenter avec un peu d'avance, et ainsi en disposer à loisir :

```
if (!Number.isNaN) {
    Number.isNaN = function(n) {
        return (
            typeof n === "number" && window.isnan( n )
        );
    };
}
var a = 2 / "foo";
var b = "foo";
console.log(Number.isNaN( a )); // true
console.log(Number.isNaN( b )); // false -- ça marche !!!
```

Pour tester si une valeur est un "entier", on peut utiliser la fonction `isInteger()` qui est prévue pour être intégrée à l'objet Number, dans la spécification de la norme ECMAScript 6.

La fonction `isInteger()` fonctionnera de la façon suivante :

```
Number.isInteger( 42 ); // true
Number.isInteger( 42.000 ); // true
Number.isInteger( 42.3 ); // false
```

Pour pouvoir bénéficier de cette fonction sur des navigateurs ne la supportant pas encore, on peut recourir au polyfill suivant :

```
if (!Number.isInteger) {
    Number.isInteger = function(num) {
        return typeof num == "number" && num % 1 == 0;
    };
}
```

La norme ECMAScript 6 prévoit même d'intégrer une fonction `isSafeInteger()` à l'objet `Number`.

```
Number.isSafeInteger( Number.MAX_SAFE_INTEGER ); // true
Number.isSafeInteger( Math.pow( 2, 53 ) ); // false
Number.isSafeInteger( Math.pow( 2, 53 ) - 1 ); // true
```

Voici un polyfill permettant de l'implémenter dès maintenant :

```
if (!Number.isSafeInteger) {
    Number.isSafeInteger = function(num) {
        return Number.isInteger( num ) && Math.abs( num ) <=
            Number.MAX_SAFE_INTEGER;
    };
}
```

3.2.4.2 Math

Contrairement à l'objet Number, l'objet Math n'a pas de constructeur. Ces propriétés et méthodes sont donc statiques.

Si on tente d'instancier un objet Math avec "new", on obtiendra une erreur.

Exemple d'utilisation :

```
var topValue = Math.max(firstValue, secondValue); // renvoie la valeur la plus grande
```

Liste des méthodes attachées à l'objet Math :

abs (x)	Retourne la valeur absolue de x, si x est NaN, renvoie NaN
acos (x)	Renvoie l'arc cosinus de x, si x est supérieur à 1 ou inférieur à 0, renvoie NaN
asin (x)	Retourne arc sinus de x, si x est supérieur à 1 ou inférieur à -1, renvoie NaN
atan (x)	Retourne l'arc tangente de x
atan2 (x, y)	Renvoie l'arc tangente du quotient de x, y
ceil (x)	Renvoie le plus petit nombre entier égal ou supérieur à x
cos (x)	Retourne le cosinus de x
exp (x)	Retourne Ex où E est la base des logarithmes naturels
floor (x)	Retourne le plus grand entier inférieur ou égal à x
log (x)	Retourne le logarithme de x
max (x1, x2, ..., xn)	Retourne le plus grand des paramètres transmis
min (x1, x2, ..., xn)	Retourne le plus petit des paramètres transmis
pow (x, y)	Retourne le résultat de l'élévation à la puissance x de y
random ()	Retourne un nombre aléatoire supérieur ou égal à 0, et moins de 1
round (x)	Arrondit au nombre entier le plus proche de la valeur transmise
sin (x)	Renvoie le sinus de x
sqrt (x)	Renvoie la racine carrée de x

Liste des propriétés attachées à l'objet Math :

E	La valeur numérique correspondant à e, la base des logarithmes népériens
LN2	logarithme naturel de 2
LN10	logarithme naturel de 10
LOG2E	logarithme en base 2 de e, et l'inverse de LN2
LOG10E	logarithme de base 10 e, et la réciproque de LN10
PI	Le nombre π
SQRT1_2	Racine carrée de 1/2, réciproque de SQRT2

SQRT2

Racine carrée de 2

3.2.4.3 Précision des calculs

PHP et Javascript ont en commun un problème de précision, en ce qui concerne les calculs de valeurs numériques comportant des parties décimales.

Par exemple : la soustraction des valeurs 35,04 et 35, devrait en toute logique renvoyer la valeur 0,04. Mais il n'en est rien, car le résultat renvoyé par cette simple soustraction est le suivant :

- en PHP : 0.03999999999999915
- en Javascript : 0.03999999999999915

Cette erreur de précision est dûe au fait que les langages PHP et Javascript s'appuient tous deux sur la norme IEEE 754 pour la représentation interne des nombres.

La documentation de PHP est assez explicite sur le problème (cf. extrait ci-dessous) :

Les nombres décimaux ont une précision limitée. Même s'ils dépendent du système, PHP utilise le format de précision des décimaux IEEE 754, qui donnera une erreur maximale relative de l'ordre de 1.11e-16 (dûe aux arrondis). Les opérations arithmétiques non-élémentaires peuvent donner des erreurs plus importantes et bien sûr les erreurs doivent être prises en compte lorsque plusieurs opérations sont liées.

*Aussi, les nombres rationnels exactement représentables sous forme de nombre à virgule flottante en base 10, comme 0.1 ou 0.7, n'ont pas de représentation exacte comme nombres à virgule flottante en base 2, utilisée en interne, et ce quelle que soit la taille de la mantisse. De ce fait, ils ne peuvent être convertis sans une petite perte de précision. Ceci peut mener à des résultats confus: par exemple, floor((0.1+0.7)*10) retournera normalement 7 au lieu de 8 attendu, car la représentation interne sera quelque chose comme 7.999999999999991118.... Ainsi, ne faites jamais confiance aux derniers chiffres d'un nombre décimal, mais aussi, ne comparez pas l'égalité de 2 nombres décimaux directement.* (Source de l'extrait : <http://php.net/manual/fr/language.types.float.php>)

Dans ces conditions, comment faire pour pallier ce problème de précision, et obtenir des résultats corrects ?

Les langages PHP et Javascript étant souvent utilisés conjointement, il convient d'étudier les solutions possibles pour les deux langages.

Le problème vu plus haut n'est pas spécifique à la soustraction, on le retrouve également sur les additions. Voici quelques exemples de calculs et les valeurs obtenues en PHP et en Javascript :

```
$x = 35.04 ;
$y = 35 ;

echo 'x - y = ' , $x - $y ;
// devrait renvoyer 0.04 mais
// renvoie 0.0399999999999999 en PHP
// renvoie 0.0399999999999915 en Javascript

echo 'y - x = ' , $y - $x ;
// devrait renvoyer -0.04 mais
// renvoie -0.0399999999999999 en PHP
// renvoie -0.0399999999999915 en Javascript

echo 'x + y = ' , $x + $y ;
// renvoie 70.04 en PHP, ce qui est le résultat attendu mais
// renvoie 70.03999999999999 en Javascript

echo 'x * y = ' , $x * $y ;
// renvoie 1226.4 en PHP, ce qui est correct, mais
// renvoie 1226.399999999999 en Javascript

echo 'x / y = ' , $x / $y ;
// renvoie 1.0011428571429 en PHP
// renvoie 1.0011428571428571 en Javascript
// renvoie 1.00114285714286 sous Excel
// renvoie 1.00114285714285714285714 avec SQL DB2 for i
```

Une première approche couramment pratiquée - qui fonctionne avec les 2 langages - consiste à gommer temporairement la notion de décimale sur les 2 facteurs de l'opération, avant d'effectuer cette opération, puis à rétablir après coup la notion de décimale.

Dans le cas de la soustraction de 35,04 par 35, cela consisterait à multiplier au préalable les 2 facteurs par 100, puis à les soustraire, et enfin à diviser le résultat de la soustraction par 100 pour récupérer la bonne valeur.

```
echo '(x*100 - y*100) / 100 = ' , ($x * 100 - $y * 100) / 100 ; // renvoie 0.04 en
PHP et en Javascript
```

Pour éviter d'avoir à effectuer ces manipulations fastidieuses pour chaque calcul, on peut écrire un objet Javascript regroupant un jeu de méthodes de calcul, comme dans l'exemple ci-dessous, fourni ici à titre d'exemple :

```
var myCalcFunctions = {} ;
myCalcFunctions.sub = function (val1, val2, precision_dec) {
    precision_dec = parseInt(precision_dec) ;
    if (precision_dec < 0 || precision_dec > 9) {
        return false ;
```

```

        }
        if (parseInt(val1) == val1 && parseInt(val2) == val2) {
            return val1 - val2 ;
        } else {
            var internal_prec = 1 ;
            for (var i = 0 ; i < precision_dec ; i += 1) {
                internal_prec *= 10 ;
            }
            var internal_val1 = (val1 * internal_prec ) ;
            var internal_val2 = (val2 * internal_prec ) ;
            return (internal_val1.toFixed() - internal_val2.toFixed()) / internal_prec ;
        }
    } ;
myCalcFunctions.add = function (val1, val2, precision_dec) {
    precision_dec = parseInt(precision_dec) ;
    if (precision_dec < 0 || precision_dec > 9) {
        return false ;
    }
    if (parseInt(val1) == val1 && parseInt(val2) == val2) {
        return val1 + val2 ;
    } else {
        var internal_prec = 1 ;
        for (var i = 0 ; i < precision_dec ; i += 1) {
            internal_prec *= 10 ;
        }
        var internal_val1 = (val1 * internal_prec ) ;
        var internal_val2 = (val2 * internal_prec ) ;
        return (parseInt(internal_val1) + parseInt(internal_val2)) / internal_prec ;
    }
} ;

var i ;
for (i = -1 ; i <= 10 ; i += 1) {
    console.log('sub. '+ i + ' => ' + myCalcFunctions.sub(x, y, i) ) ;
    // console.log(x + ' - ' + y + ' = ' + (x - y));
}
for (i = -1 ; i <= 10 ; i += 1) {
    console.log('add. '+ i + ' => ' + myCalcFunctions.add(x, y, i) ) ;
    // console.log(x + ' + ' + y + ' = ' + (x + y));
}

//=> résultats ci-dessous obtenus via le jeu de tests ci-dessus :
sub. -1 =>
sub. 0 => 0
sub. 1 => 0
sub. 2 => 0.04
sub. 3 => 0.04
sub. 4 => 0.04
sub. 5 => 0.04
sub. 6 => 0.04
sub. 7 => 0.04

```

```

sub. 8 => 0.04
sub. 9 => 0.04
sub. 10 =>
add. -1 =>
add. 0 => 70
add. 1 => 70
add. 2 => 70.04
add. 3 => 70.04
add. 4 => 70.04
add. 5 => 70.04
add. 6 => 70.04
add. 7 => 70.04
add. 8 => 70.04
add. 9 => 70.04
add. 10 =>

```

Il est possible d'implémenter en PHP une classe abstraite regroupant un jeu de méthodes similaires à ce qui est implémenté ci-dessus en Javascript (un exemple de code peut être fourni sur simple demande aux lecteurs intéressés).

En PHP, une seconde approche consiste à influer sur le niveau de précision interne utilisé par PHP pour les calculs.

Par défaut, ce niveau de précision est généralement fixé à 14 dans le fichier de configuration php.ini (cf. extrait ci-dessous) :

```

; The number of significant digits displayed in floating point numbers.
; http://php.net/precision
precision = 14

```

Le fait d'augmenter la précision en la passant par exemple à 17, comme on peut le lire dans certains forums, ne règle pas le problème, il ne fait que l'aggraver.

En fixant la précision à 16, on est très proche du niveau de précision de Javascript.

Avec cette précision de 16, voici les valeurs renvoyées par PHP :

```

x = 35.04
y = 35
x - y = 0.03999999999999915
y - x = -0.03999999999999915
x + y = 70.03999999999999
x * y = 1226.4
x / y = 1.001142857142857
(x*1000 - y*1000) / 1000 = 0.04

```

Et voici les valeurs renvoyées par Javascript pour les mêmes valeurs de x et y :

```

x - y = 0.03999999999999915
y - x = -0.03999999999999915
x + y = 70.03999999999999

```

```
x * y = 1226.399999999999
x / y = 1.0011428571428571
(x*1000 - y*1000) / 1000 = 0.04
```

On voit que certains écarts subsistent selon le type de calcul, entre les 2 langages (et que de toute façon, la plupart des calculs sont faux).

Si l'on réduit le niveau de précision de PHP à 9 décimales, la précision des calculs s'améliore très nettement :

```
x - y = 0.04
y - x = -0.04
x + y = 70.04
x * y = 1226.4
x / y = 1.00114286
(x*1000 - y*1000) / 1000 = 0.04
```

Il est donc possible, et peut être souhaitable, de modifier le fichier de configuration PHP.ini, en ramenant la précision à une valeur comprise entre 12 et 9 décimales (valeurs à tester en fonction des calculs à effectuer).

ATTENTION : si l'on ne souhaite pas modifier le fichier PHP.ini, il est possible d'influer sur la précision des calculs pendant la durée d'exécution d'un script PHP, en utilisant l'instruction suivante :

```
ini_set('precision', 9);
```

Côté Javascript, la solution passe plutôt par l'utilisation de la méthode `toPrecision()` qui s'applique aux variables numériques (cf. le lien ci-dessous pour la documentation) :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Number/toPrecision

En appliquant la fonction « maison » ci-dessous (fonction utilisant la méthode `toPrecision()` avec une précision de 12 décimales), au résultat des différents calculs vus dans le jeu d'essai précédent :

```
var decPrecision = function (number) {
    var tmp = parseFloat(number) ;
    return (tmp.toPrecision(12));
}

var x = 35.04 ;
var y = 35 ;

var calc1 = x - y ;
```

```
console.log( 'x - y = ', decPrecision(calc1) ) ;  
  
var calc2 = y - x ;  
console.log( 'y - x = ' , decPrecision(calc2) );  
  
var calc3 = x + y ;  
console.log( 'x + y = ', decPrecision(calc3) );  
  
var calc4 = x * y ;  
console.log( 'x * y = ', decPrecision(calc4) );  
  
var calc5 = x / y ;  
console.log( 'x / y = ', decPrecision(calc5) );  
  
var calc6 = (x * 1000 - y * 1000) / 1000 ;  
console.log( '(x*1000 - y*1000) / 1000 = ' , decPrecision(calc6) );
```

... on obtient en sortie les résultats suivants :

```
x - y =  0.0400000000000  
y - x = -0.0400000000000  
x + y =  70.0400000000  
x * y =  1226.40000000  
x / y =  1.00114285714  
(x*1000 - y*1000) / 1000 =  0.0400000000000
```

En fixant la valeur du paramètre de la fonction toPrecision() entre 9 et 12 décimales, on obtient de bons résultats, mais il convient là encore de procéder à des tests en fonction des valeurs traitées par l'application considérée.

3.2.5 Dates and Times

La valeur numérique renvoyée par l'objet Date correspond au nombre de secondes depuis le 1er janvier 1970 (temps UTC).

Méthodes get de l'objet Date :

Method	Purpose
getDate	Returns day of the month (0–31)
getDay	Returns day of the week (0–6)
getFullYear	Returns 4-digit full year
getHours	Returns local hour (0–23)
getMilliseconds	Returns local milliseconds (0–999)
getMinutes	Returns local minute (0–59)
getMonth	Returns local month (0–11)
getSeconds	Returns local second (0–59)
getTime	Returns number of seconds since January 1, 1970 00:00:00 UTC
getTimezoneOffset	Returns time zone from UTC
getUTCDate	Returns day of month in UTC time (0–31) method (Date)
getUTCDay	Returns day of the week in UTC time (0–6)
getUTCFullYear	Returns 4-digit UTC year
getUTCHours	Returns UTC hours (0–23)
getUTCMilliseconds	Returns UTC milliseconds (0–999)
getUTCMinutes	Returns UTC minutes (0–59)
getUTCMonth	Returns UTC month (0–11)
getUTCSeconds	Returns UTC seconds (0–59)

Méthodes set de l'objet Date :

Method	Purpose
setDate	Sets the day of month (1–31)
setFullYear	Sets 4-digit full year
setHours	Sets the hour (0–23)
setMilliseconds	Sets the date's milliseconds (0–999)
setMinutes	Sets the date's minutes (0–59)
setMonth	Sets the month (0–11)
setSeconds	Sets the seconds (0–59)
setTime	Sets the date's time as milliseconds since January 1, 1970 00:00:00 UTC
setUTCDate	Sets the date's day of month in UTC

setUTCFullYear	Sets the full year in UTC
setUTCHours	Sets the date's hours in UTC
setUTCMilliseconds	Sets the date's milliseconds in UTC
setUTCMilliseconds	Sets the date's minutes in UTC
setUTCMonth	Sets the month in UTC
setUTCSeconds	Sets the seconds in UTC

Construction d'une date à partir de la date système :

```
console.log(new Date()); // Date {Mon Nov 03 2014 16:51:55 GMT+0100}
console.log(Date()); // Mon Nov 03 2014 16:55:28 GMT+0100
```

On constate que la fonction Date() renvoie une chaîne, alors que le constructeur new appliqué à cette même fonction génère un objet de type Date contenant cette même date.

Construction d'une date à partir de 3 éléments (année, mois, jour) :

```
var month = 10; // le mois 10, avec une numérotation partant de zéro,
                // correspond au mois de novembre
var day = 18;
var year = 1954;
var dt = new Date(year,month,day); // time is set to zero by default
console.log(dt); // Date {Thu Nov 18 1954 00:00:00 GMT+0100}
```

Calcul d'une date dans 10 jours à partir de la date courante :

```
var futureDate = new Date();
var info = futureDate.setDate(futureDate.getDate() + 10);
console.log(info); // 1393802525632
```

Calcul d'une date à partir de l'année courante - 18 ans :

```
var pastDate = new Date();
var result = pastDate.setFullYear(pastDate.getFullYear() - 18);
console.log(result); // 824858680476
```

Autres exemples avec les méthodes get :

```
var dt = new Date();
var month = dt.getMonth(); // 1 si février
var day = dt.getDate(); // 21
var year = dt.getFullYear(); // 2014
```

Construction d'une date au format ISO 8601 :

```
var dt = new Date();
// get month and increment
var mnth = dt.getUTCMonth();
mnth++;
if (mnth < 10) mnth="0" + mnth;
var day = dt.getUTCDate();
if (day < 10) day="0" + day;
var yr = dt.getUTCFullYear();
var hrs = dt.getUTCHours();
if (hrs < 10) hrs = "0" + hrs;
var min = dt.getUTCMinutes();
if (min < 10) min = "0" + min;
var secs = dt.getUTCSeconds();
if (secs < 10) secs = "0" + secs;
var newdate = yr + "-" + mnth + "-" + day + "T" + hrs + ":" + min + ":" +
secs + "Z";
console.log(newdate); // 2014-02-20T23:31:07Z
```

Il peut être utile de disposer d'une méthode permettant de récupérer une date au format "timestamp Unix". La méthode now() de l'objet Date, intégrée à JS à partir de la norme ECMAScript 5, permet d'obtenir ce résultat. Pour les navigateurs anciens n'implémentant pas cette méthode, il existe un polyfill qui est le suivant :

```
if (!Date.now) {
    Date.now = function(){
        return (new Date()).getTime();
    };
}
var testdate = Date.now() ;
console.log(testdate) ; // 1415030044778
```

3.2.6 Null et Undefined

Il est très important de savoir distinguer une variable "nulle" d'une variable "undefined".

En théorie, une variable qui a été déclarée sans être initialisée renvoie la valeur undefined. Mais en pratique, certains navigateurs (comme Chrome) renvoient null dans ce cas, tandis que d'autres renvoient "undefined".

Pour éviter toute ambiguïté et tout risque d'erreur, il est recommandé d'intialiser toute variable avec une valeur par défaut (blanc, zéro, ou éventuellement null).

```
var nullvar = undefined;
console.log(nullvar); // undefined

var nullvar;
console.log(nullvar); // null pour Firefox, undefined pour Chrome et IE

var nullvar = null;
console.log(nullvar); // null
```

On peut tester si une variable est différente de null et de undefined au moyen du test suivant :

```
if (nullvar) {
    console.log("ok");
} else {
    console.log("bad");
}
```

Mais l'inconvénient de ce test, c'est qu'il renverra aussi "bad" si nullvar est égal à zéro (zéro étant considéré par Javascript comme équivalent de false).

En d'autres termes, zéro, nul, undefined, NaN (Not a Number), et la chaîne vide sont intrinsèquement faux, tout le reste est intrinsèquement vrai.

On peut souhaiter obtenir une plus grande précision dans l'analyse du contenu d'une variable. On peut dans ce cas vérifier le type de la variable,

```
/* renvoie true si la variable existe, qu'il s'agit d'une chaîne, et
   qu'elle a une longueur supérieure à zéro */
var variable = ' ';
if(((typeof variable != "undefined") &&
   (typeof variable.valueOf() == "string")) &&
   (variable.length > 0)) {
    console.log("good");
} else {
    console.log("bad");
}
```

Pour information, l'instruction "typeof" peut renvoyer l'une des valeurs lui-même :

- "number" si la variable est un nombre
- "string" si la variable est une chaîne
- "boolean" si la variable est booléen
- "function" si la variable est une fonction
- "object" si la variable est null, ou un tableau, ou encore un objet JavaScript
- "undefined" si la variable est indéfinie

3.2.6 Détection de types

Devant la difficulté à déterminer sans risque d'erreur le type de certaines données, certains frameworks JS se sont dotés d'un arsenal de fonctions pour faciliter ce type d'opération. C'est le cas du framework YUI, qui fournit un jeu très complet de méthodes pour déterminer avec précision le type de chaque donnée que l'on est susceptible de devoir manipuler en Javascript. L'exemple ci-dessous est emprunté au livre "YUI 3 Cookbook, édité chez O'Reilly".

```
<!DOCTYPE html>
<title>Checking types</title>
<script src="http://yui.yahooapis.com/3.5.0/build/yui/yui-min.js"></script>
<script>
YUI().use(function (Y) {
    var typeChecks = [
        Y.Lang.isArray([]), // => true
        Y.Lang.isArray(arguments), // => false
        Y.Lang.isArray('string'), // => false
        Y.Lang.isBoolean(true), // => true
        Y.Lang.isBoolean(0), // => false
        Y.Lang.isDate(new Date()), // => true
        Y.Lang.isDate(123), // => false
        Y.LangisFunction(function () {}), // => true
        Y.LangisFunction('test'), // => false
        Y.Lang.isNull(null), // => true
        Y.Lang.isNull(undefined), // => false
        Y.Lang.isNumber(42), // => true
        Y.Lang.isNumber('42'), // => false
        Y.Lang.isNumber(NaN), // => false
        Y.LangisObject({}), // => true
        Y.LangisObject([]), // => true (yep, arrays are objects!)
        Y.LangisObject('test'), // => false
        Y.LangisObject(function () {}), // => true
        Y.LangisObject(function () {}, true), // => false ('failFn' flag)
        Y.LangisString('string'), // => true
        Y.LangisString(42), // => false
        Y.LangisUndefined(undefined), // => true
        Y.LangisUndefined(null), // => false
        Y.LangisValue(false), // => true
        Y.LangisValue(0), // => true
        Y.LangisValue(''), // => true
        Y.LangisValue(null), // => false
        Y.LangisValue(undefined), // => false
        Y.Lang.type([]), // => array
        Y.Lang.type(true), // => boolean
        Y.Lang.type(new Date()), // => date
        Y.Lang.type(new Error()), // => error
        Y.Lang.type(function () {}), // => function
        Y.Lang.type(null), // => null
    ];
});</script>
```

```
Y.Lang.type(42), // => number
Y.Lang.type({}), // => object
Y.Lang.type(/regexp/), // => regexp
Y.Lang.type('string'), // => string
Y.Lang.type(undefined) // => undefined
];
Y.Array.each(typeChecks, function (result) {
    Y.log(result);
});
</script>
```

3.3 Structures conditionnelles

Les structures conditionnelles en Javascript sont quasi identiques à aux structures conditionnelles de PHP ou de C.

Exemple :

```
if (stateCode === '0') {
    taxe = 3.5;
} else {
    taxe = 4.5;
}
```

A noter qu'il aurait été possible d'écrire ceci :

```
if (stateCode === '0') taxe = 3.5;
else taxe = 4.5;
```

Mais les accolades ont le mérite de rendre le code plus clair et lisible.

De plus, les accolades permettent d'encapsuler plusieurs instructions, comme dans l'exemple suivant :

```
if (stateCode === '0') {
    taxe1 = 3.5;
    taxe2 = 1.5;
} else {
    taxe1 = 4.5;
    taxe2 = 2.5;
}
```

On peut également écrire des tests en cascade comme dans l'exemple suivant :

```
if (stateCode === 'OR') {
    taxPercentage = 3.5;
} else if (stateCode === 'CA') {
    taxPercentage = 5.0;
} else if (stateCode === 'MO') {
    taxPercentage = 1.0;
} else {
    taxPercentage = 2.0;
}
```

On peut préférer au code ci-dessus, le code ci-dessous s'appuyant sur la structure "switch...case", et au résultat strictement équivalent :

```
switch (stateCode) {
```

```

case 'OR':
    taxPercentage = 3.5;
    break;
case 'CA':
    taxPercentage = 5.0;
    break;
case 'MO':
    taxPercentage = 1.0;
    break;
default:
    taxPercentage = 2.0;
}

```

Autre exemple, dans lequel les statecode "CA", "NY" et "VT" ont le même effet :

```

switch (stateCode) {
case 'OR':
    taxPercentage = 3.5;
    break;
case 'MO':
    taxPercentage = 1.0;
    statePercentage = 1.5;
case 'CA':
case 'NY':
case 'VT':
    statePercentage = 2.6;
    taxPercentage = 4.5;
    break;
case 'TX':
    taxPercentage = 3.0;
    break;
default:
    taxPercentage = 2.0;
    statePercentage = 2.3;
}

```

3.4 Comparaison de contenu et de type

La comparaison de 2 valeurs se fait strictement de la même façon qu'en PHP, à savoir :

`==` renvoie true si 2 valeurs ont 2 contenus équivalents (par exemple 25 et "25"), renvoie false dans le cas contraire
`!=` renvoie true si 2 valeurs ont des contenus différents (par exemple 25 et "30"), renvoie false dans le cas contraire
`==` renvoie true si 2 valeurs ont le même contenu et sont de même type, renvoie false dans le

cas contraire

`!==` renvoie `false` si 2 valeurs ont des contenu différents, ou sont de type différent

Exemples :

```
// Comparaison du nombre 5 et de la chaîne 5
console.log(5 == "5"); // true
console.log(5 === "5"); // false

// Comparaison du nombre 5 avec la valeur équivalente exprimée en hexadécimal
console.log(25 == "0x19"); // true
console.log(25 === "0x19"); // false

// Comparaison du nombre 1 avec true
console.log(1 == true); // true
console.log(1 === true); // false

// Comparaison du nombre 0 avec false
console.log(0 == false); // true
console.log(0 === false); // false

// Comparaison du nombre 2 avec true
console.log(2 == true); // false
console.log(2 === true); // false

// Comparaison d'un objet renvoyant l'équivalent hexadécimal de 25 avec la valeur 25
var object = {
    toString: function() {
        return "0x19";
    }
};
console.log(object == 25); // true
console.log(object === 25); // false

// Null et undefined
console.log(null == undefined); // true
console.log(null === undefined); // false
```

3.5 Constantes

La plupart des langages proposent la notion de constante, mais Javascript fait exception à la règle.

Il existe cependant une implémentation de constante avec l'instruction "const" :

```
const CURRENT_MONTH = 3 ;
```

L'implémentation actuelle de const est une extension spécifique à Mozilla et ne fait pas partie de la norme ECMAScript 5. Cette implémentation est prise en charge dans Firefox et Chrome (V8), de même que sur certaines versions de Safari et d'Opera. Il est envisagé que l'instruction "const" soit intégrée à la norme ECMAScript 6 (en préparation), mais pour l'heure, on recommandera de déclarer les constantes comme des variables, mais en les codifiant en majuscule de manière à pouvoir les distinguer des "vraies" variables :

```
var CURRENT_MONTH = 3 ;
```

3.6 Tableaux

3.6.1 Introduction

Les tableaux en Javascript sont des regroupements de valeurs indexés à partir de zéro.

Les tableaux peuvent être créés au moyen de la fonction constructeur Array(), mais ils peuvent également déclarés sous forme littérale :

```
// Un tableau créé au moyen du constructeur Array()
var a = new Array("produit1", "produit2", "produit3");

// Le même tableau créé de manière littérale
var a = ["produit1", "produit2", "produit3"];
console.log(typeof a); // "object", parce que les tableaux sont des objets
console.log(a.constructor === Array); // true
```

Il semble se dégager un consensus au sein de la communauté des développeurs Javascript, pour utiliser la déclaration littérale, de préférence à la déclaration via le constructeur Array().

Il y a en effet un piège dans l'utilisation du constructeur Array(). Si on ne lui transmet qu'un seul paramètre, de type numérique, alors ce paramètre va avoir pour effet de créer un tableau dont le nombre de postes correspond à la valeur de ce paramètre (le paramètre n'est pas conservé pour alimenter le premier poste du tableau). Ce n'est qu'à partir de 2 paramètres, que le constructeur Array() les considère comme des valeurs devant être stockées dans un tableau (et crée le tableau correspondant). Cela peut être source de confusion, et donc d'erreur.

Après avoir initialisé notre tableau « a », on peut ajouter de nouveaux éléments à la volée, au moyen de la méthode « push » :

```
a.push("produit4") ;  
a.push("produit5") ;
```

On peut connaître la longueur d'un tableau au moyen de la méthode « length » :

```
console.log(a.length()) ; // renverra la valeur 5
```

On peut écrire une boucle qui parcourt l'ensemble des postes d'un tableau :

```
> for (var i = 0, imax = a.length ; i < imax ; i++) {  
    console.log(a[i]);  
}  
produit1  
produit2  
produit3  
produit4  
produits  
< undefined
```

VM7385:2
VM7385:2
VM7385:2
VM7385:2
VM7385:2

On peut parcourir le tableau dans l'autre sens :

```
> for (var i = a.length - 1 ; i >= 0 ; i--) {  
    console.log(a[i]);  
}  
produits  
produit4  
produit3  
produit2  
produit1  
< undefined
```

VM10315:2
VM10315:2
VM10315:2
VM10315:2
VM10315:2

On peut aussi parcourir le tableau avec une boucle « while » :

```
> var i = 0;
  var imax = a.length;
  while (i < imax) {
    console.log(a[i]);
    i++;
  }
produit1                                     VM31076:4
produit2                                     VM31076:4
produit3                                     VM31076:4
produit4                                     VM31076:4
produit5                                     VM31076:4
< 4
```

La méthode « foreach » constitue un autre moyen très pratique de parcourir un tableau :

```
> a.forEach(function(element, index) {
  console.log(index + ' ' + element  );
}) ;
0 produit1                                     VM18339:2
1 produit2                                     VM18339:2
2 produit3                                     VM18339:2
3 produit4                                     VM18339:2
4 produit5                                     VM18339:2
< undefined
```

A noter : la méthode "forEach" est intégrée à la norme ECMAScript 5. Elle est bien supportée par la plupart des navigateurs, à l'exclusion des versions d'IE antérieures à la 9.

Pour les navigateurs ne supportant pas cette méthode, Mozilla propose un polyfill :

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

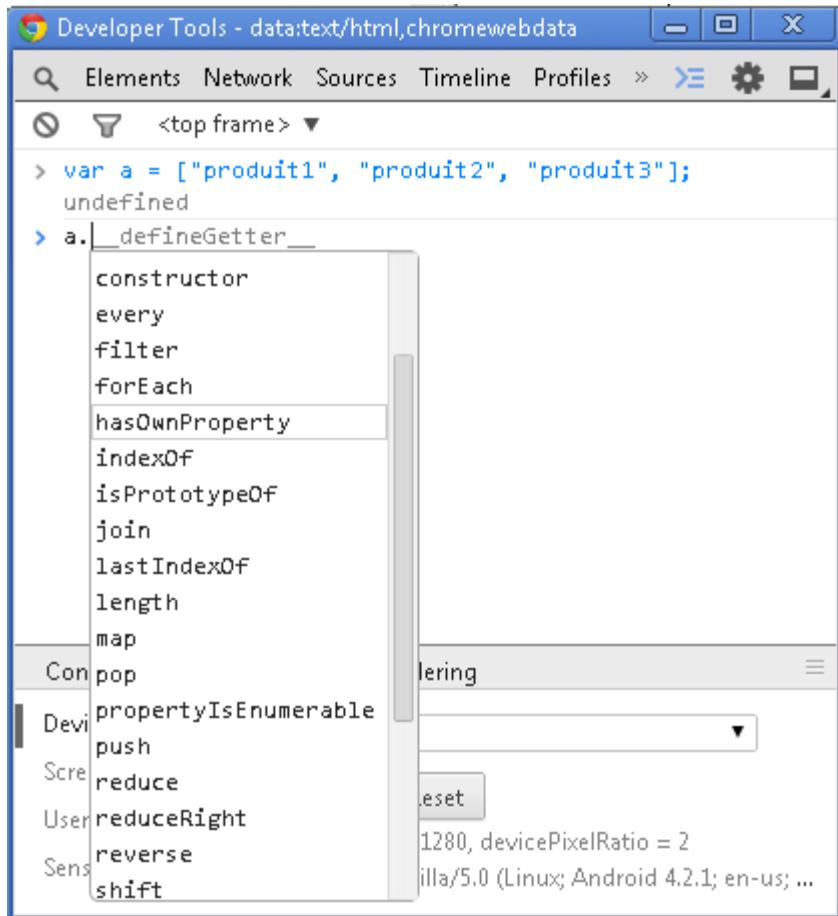
```
if (!Array.prototype.forEach) {  
    Array.prototype.forEach = function(fun /*, thisArg */){  
        "use strict";  
  
        if (this === void 0 || this === null) throw new TypeError();  
  
        var t = Object(this);  
        var len = t.length >>> 0;  
        if (typeof fun !== "function") throw new TypeError();  
  
        var thisArg = arguments.length >= 2 ? arguments[1] : void 0;  
        for ( var i = 0; i < len; i++) {  
            if (i in t) fun.call(thisArg, t[i], i, t);  
        }  
    };  
}
```

POINT IMPORTANT : La fonction forEach() s'applique aux tableaux Javascript, mais pas aux "Node Lists". Mais on verra, dans le chapitre consacré aux "Node Lists", qu'il est possible de contourner cette limitation, en utilisant une technique particulière, que nous ne détaillerons pas maintenant.

Il peut être intéressant de disposer d'une méthode isArray() permettant de déterminer si une variable est bien un tableau. Cette méthode existe sous ECMAScript5, mais elle n'existe pas sur les versions antérieures de la norme. On peut l'émuler sur les navigateurs qui en sont dépourvus au moyen du code suivant :

```
if (typeof Array.isArray === "undefined") {  
    Array.isArray = function (arg) {  
        return Object.prototype.toString.call(arg) === "[object Array]";  
    };  
}
```

L'autocomplétion de Google Chrome nous permet de prendre connaissance rapidement des différentes méthodes attachées à l'objet Array :



On va voir dans les chapitres suivants quelques une de ces méthodes, parmi les plus utilisées.

3.6.2 Trier un tableau

Le tri d'un tableau se fait au moyen des méthodes `sort()` et `reverse()`.

Exemple :

```
var numberArray = [4,13,2,31,5];
var tableau_tri_asc = numberArray.sort();
console.log(tableau_tri_asc); // [13, 2, 31, 4, 5]
var tableau_tri_desc = numberArray.reverse();
```

```
console.log(tableau_tri_desc); [5, 4, 31, 2, 13]
```

On constate que le tri obtenu est un tri alphabétique, ce qui ne correspond pas forcément au résultat attendu.

Mais on peut modifier ce comportement, via un paramètre optionnel de la méthode `sort()`, auquel on peut fournir une fonction de tri personnalisée :

```
function compareNumbers(a, b) {
    return a > b;
}
var numArray = [13,2,31,4,5];
var tableau_tri_num = numArray.sort(compareNumbers);
console.log(tableau_tri_num); // [2, 4, 5, 13, 31]
```

A noter que l'on n'est pas obligé de déclarer une fonction `compareNumbers`, une fonction anonyme peut suffire dans certains cas :

```
var numArray = [13,2,31,4,5];
var tableau_tri_num = numArray.sort(function (a, b) {
    return a > b;
});
console.log(tableau_tri_num); // [2, 4, 5, 13, 31]
```

3.6.3 Tableau multidimensionnel

La création de tableau multi-dimensionnel ne pose pas de difficulté particulière.
La solution consiste à créer des tableaux imbriqués.

Exemples :

```
// Définition de la longueur du tableau
var arrayLength = 3, i=0;
// Création du tableau
var multiArray = new Array(arrayLength);
for (var i = 0; i < arrayLength; i+=1) {
    multiArray[i] = new Array(arrayLength);
}
// Ajout d'éléments au premier poste du tableau
multiArray[0][0] = "pomme";
multiArray[0][1] = "banane";
multiArray[0][2] = "cerise";
```

```
// Ajout d'éléments au premier second du tableau
multiArray[1][0] = 2;
multiArray[1][1] = 56;
multiArray[1][2] = 83;

// Ajout d'éléments au troisième poste du tableau
multiArray[2][0] = ['baleine','mer'];
multiArray[2][1] = ['singe','arbre'];
multiArray[2][2] = ['mars','espace'];

console.log(multiArray); // [[["pomme", "banane", "cerise"], [2, 56, 83], [["baleine", "mer"], ["singe", "arbre"], ["mars", "espace"]]]]
console.log(multiArray[1]); // sous-tableau [2, 56, 83]
console.log(multiArray[2][2][0]); // mars
```

3.6.4 Dupliquer des portions de tableaux

La méthode `slice()` permet de créer un nouveau tableau à partir d'une portion de tableau existant.

Elle retourne les éléments sélectionnés dans un tableau, sous forme d'un nouvel objet tableau, sans toucher au tableau d'origine.

Elle sélectionne les éléments à partir de la position précisée dans le premier paramètre de la méthode, et elle poursuit la recherche jusqu'à la position précisée dans le second paramètre, sans retenir la valeur correspondant à la dernière position.

Exemple :

```
var fruits = ["Banane", "Orange", "Citron", "Pomme", "Mangue"];
var fruits2 = fruits.slice(1,3);
console.log(fruits2) ; // ["Orange", "Citron"]
```

Quelques précisions :

- on peut omettre le second paramètre qui est optionnel et ainsi sélectionner tous les postes du tableau d'origine, à partir du point de départ précisé dans le premier paramètre
- on peut indiquer des valeurs négatives pour les 2 paramètres, et ainsi effectuer la sélection à partir de la fin du tableau d'origine
- si les éléments copiés sont des valeurs littérales, comme des chaînes, des nombres ou des valeurs booléennes, alors ils sont copié par valeur (donc modifier après coup une valeur dans le tableau d'origine n'a pas d'incidence sur le tableau de destination)
- si les éléments copiés sont des objets, alors ils sont copiés par référence (toute modification effectuée après coup dans le tableau d'origine se répercute automatiquement dans le tableau

de destination)
 - la méthode slice() n'est pas supportée par IE8

3.6.5 Recherche dans un tableau

La norme ECMAScript 5 apporte de nouvelles méthodes pour faciliter la recherche d'informations à l'intérieur de tableaux.

Ce sont les méthodes indexOf() and lastIndexOf().

En réalité, elles existaient déjà avant la norme ECMAScript 5, et la norme n'a fait que les intégrer et les standardiser. C'est ce qui explique qu'elles soient plutôt bien supportées par la plupart des navigateurs (à l'exception d'IE 8).

Exemple :

```
var animaux = new Array("chien", "chat", "poisson", "éléphant", "perroquet", "lion");
console.log(animaux.indexOf("éléphant")); // 3
console.log(animaux.lastIndexOf("elephant")); // -1 (pas trouvé)
console.log(animaux.lastIndexOf("éléphant")); // 3 (dernière occurrence
correspondant à la recherche)
```

La documentation Mozilla propose une solution pour émuler la méthode indexOf sur IE 8 :

https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Array/indexOf

```
if (!Array.prototype.indexOf) {
    Array.prototype.indexOf = function(elt /*, from*/) {
        var len = this.length >>> 0;
        var from = Number(arguments[1]) || 0;
        from = (from < 0) ? Math.ceil(from) : Math.floor(from);
        if (from < 0)
            from += len;
        for (; from < len; from++) {
            if (from in this && this[from] === elt)
                return from;
        }
        return -1;
    };
}
```

3.6.6 Recherche et remplacement

Pour remplacer des éléments dans un tableau, on peut combiner les méthodes `indexOf()`, ou `lastIndexOf()`, avec la méthode `splice()`.

On rappelle que ces méthodes ne sont pas supportées par IE8.

Exemple :

```
var animaux = ["chien", "chat", "phoque", "pingouin", "lion", "chat"];
console.log(animaux); // ["chien", "chat", "pingouin", "lion", "singe"]

// retirer l'élément "phoque" du tableau
animaux.splice(animaux.indexOf("phoque"),1); // ["chien", "chat", "phoque",
"pingouin", "lion"]
console.log(animaux);

// remplacer le dernier élément "chat" par "singe"
animaux.splice(animaux.lastIndexOf("chat"),1,"singe"); // ["chien", "chat", "phoque",
"pingouin", "lion"]
console.log(animaux);

// remplacer le dernier élément "lion" par "zèbre", et ajouter dans la foulée
// "éléphant"
animaux.splice(4, 1, "zèbre", "éléphant");
console.log(animaux); // ["chien", "chat", "phoque", "pingouin", "zèbre",
"éléphant"]
```

Autre exemple de manipulation :

```
var charSets = ["ab", "bb", "cd", "ab", "cc", "ab", "dd", "ab"];
console.log(charSets); // ["ab", "bb", "cd", "ab", "cc", "ab", "dd", "ab"]
// remplacement de tous les "ab" par "***"
while (charSets.indexOf("ab") != -1) {
    charSets.splice(charSets.indexOf("ab"),1,"***");
}
console.log(charSets); // [***, "bb", "cd", ***, "cc", ***, "dd", ***]
// suppression de tous les éléments ***
while(charSets.indexOf("**") != -1) {
    charSets.splice(charSets.indexOf("**"),1);
}
console.log(charSets); // ["bb", "cd", "cc", "dd"]
```

3.6.7 Appliquer une fonction à un tableau

On peut souhaiter appliquer une fonction à chacun des éléments d'un tableau.

Pour ce faire, on peut utiliser la méthode "forEach", en lui passant en paramètre une fonction de rappel (en anglais : "callback").

Premier exemple :

```
var charSets = ["ab", "bb", "cd", "ab", "cc", "ab", "dd", "ab"];
function replaceElement(element, index, array) {
    if (element == "ab") {
        array[index] = "***";
    }
}
// Application de la fonction replaceElement à chacun des postes du tableau charSets
charSets.forEach(replaceElement);
console.log(charSets); // [ "***", "bb", "cd", "***", "cc", "***", "dd", "***" ]
```

Second exemple (plus complet) : soit le tableau numérique processData2 contenant 3 objets. Les 3 boucles ci-dessous produisent strictement le même résultat :

```
var processData2 = [];
processData2.push ({ name:'toto0' }) ;
processData2.push ({ name:'toto1' }) ;
processData2.push ({ name:'toto2' }) ;

function printBr(element, index, array) {
    console.log(index + ' ' + element.name );
}

console.log('boucle1') ;
processData2.forEach(printBr);

console.log('boucle2') ;
[].forEach.call(processData2, printBr);

console.log('boucle3') ;
[].forEach.call(processData2, function(element, index) {
    console.log(index + ' ' + element.name );
});
```

ATTENTION : la méthode "forEach" ne supporte pas les tableaux associatifs. Les 3 boucles ci-dessus ne fonctionneraient donc pas si le tableau processData2 avait été initialisé de la façon suivante :

```
var processData2 = [];
processData2[ 'objet0' ] = { name:'toto0' } ;
processData2[ 'objet1' ] = { name:'toto1' } ;
processData2[ 'objet2' ] = { name:'toto2' } ;
```

3.6.8 Tableau associatif

Le langage Javascript ne fournit pas de structure de type "tableau associatif", comme on peut en trouver dans d'autres langages. Mais les objets constituent une méthode de substitution pratique, et puissante.

Exemple :

```
var arrayAssoc = {
    myString: 'string',
    myNumber: 10,
    myBoolean: false,
    myNull: null,
    myUndefined: undefined
};
```

Pour manipuler notre tableau associatif, on dispose de plusieurs syntaxes équivalentes :

- pour récupérer les valeurs du tableau :

```
var myStr = arrayAssoc.myString ;
var myStr = arrayAssoc['myString'] ;
```

- pour modifier les valeurs du tableau :

```
arrayAssoc.myString = 'bonjour' ;
arrayAssoc['myString'] = 'bonjour' ;
```

On peut aussi ajouter au tableau associatif de nouvelles propriétés de manière dynamique :

```
arrayAssoc.myString2 = 'au revoir' ;
arrayAssoc['myString2'] = 'au revoir' ;
```

Exemple de formulaire HTML permettant de "jouer" avec la notion de tableau associatif :

Valeur 1:	10
Valeur 2:	20
Valeur 3:	30
Valeur 4:	
<input type="button" value="Valider"/>	

Le code source :

```
<!DOCTYPE html>
<html lang="fr">
```

```

<meta charset="utf-8">
<head>
<title>Tableau associatif</title>
<script>
//<![CDATA[
/*
 * prend les champs du formulaire et stocke les valeurs dans un tableau
 * associatif (en fait un objet)
 */
function getVals(evt) {
    var parent_form = this.parentNode ;
    var elems = parent_form.elements;
    var elemArray = new Object();
    var str = '' ;
    var i, i_len ;

    for ( i = 0, i_len=elems.length; i < i_len ; i+=1) {
        if (elems[i].type == "text")
            elemArray[elems[i].id] = elems[i].value;
    }

    str = checkVals(elemArray);
    document.getElementById("result").innerHTML = str;

    console.log(str) ; // first=10;second=20;third=40;four=;
    console.log(elemArray); /* Object { first="10", second="20",
        third="40", more...} */
    console.log(JSON.stringify( elemArray)) ; /* format JSON :
        {"first":"10","second":"20","third":"30","four":""} */

    this.preventDefault; /* stopper les actions par défaut déclenchées
        par l'évènement */
    this.stopPropagation; /* stopper la propagation de l'évènement
        aux noeuds parents */
    return false;
}

/*
 * Lit le tableau associatif reçu en paramètre et le renvoie sous
 * la forme d'une chaîne
 */
function checkVals(elemArray) {
    var str = "";
    var key ;
    for ( key in elemArray) {
        str += key + "=" + elemArray[key] + ";";
    }
    return str ;
}
//--><!]>
</script>
</head>

```

```

<body>
    <form>
        <label>Valeur 1:</label>
        <input type="text" id="first" /><br />
        <label>Valeur 2:</label>
        <input type="text" id="second" /><br />
        <label>Valeur 3:</label>
        <input type="text" id="third" /><br />
        <label>Valeur 4:</label>
        <input type="text" id="four" /><br />
        <input id="picker" type="button" value="Valider" />
    </form>
    <div id="result"></div>

<script>
//<![CDATA[
    (function(){
        var sub_picker = document.getElementById('picker');
        sub_picker.addEventListener('click', getVals, false);
    })() ;
//--><!]>
</script>
</body>
</html>

```

3.6.9 Méthodes de l'objet Array

A chaque évolution de la norme EcmaScript, l'objet Javascript Array (Tableau) s'enrichit de nouvelles méthodes rendant la manipulation de listes plus facile.

La page de référence de Mozilla permet d'obtenir un aperçu des méthodes disponibles :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/

- Array.of()
- Array.prototype.concat()
- Array.prototype.copyWithin()
- Array.prototype.entries()
- Array.prototype.every()
- Array.prototype.fill()
- Array.prototype.filter()
- Array.prototype.find()
- Array.prototype.findIndex()
- Array.prototype.forEach()

- `Array.prototype.includes()`
- `Array.prototype.indexOf()`
- `Array.prototype.join()`
- `Array.prototype.keys()`
- `Array.prototype.lastIndexOf()`
- `Array.prototype.map()`
- `Array.prototype.pop()`
- `Array.prototype.push()`
- `Array.prototype.reduce()`
- `Array.prototype.reduceRight()`
- `Array.prototype.reverse()`
- `Array.prototype.shift()`
- `Array.prototype.slice()`
- `Array.prototype.some()`
- `Array.prototype.sort()`
- `Array.prototype.splice()`
- `Array.prototype.toLocaleString()`
- `Array.prototype.toSource()`
- `Array.prototype.toString()`
- `Array.prototype.unshift()`
- `Array.prototype.values()`
- `Array.prototype[@@iterator]()`

Sachant que certaines de ces méthodes ne sont disponibles qu'à partir de ES5 ou ES6, le site de Mozilla propose des polyfill pour la plupart de ces méthodes.

Petite sélection de méthodes qui peuvent être utiles au quotidien :

- La méthode `reduce()` applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.
- La méthode `values()` renvoie un nouvel objet Array Iterator qui contient les valeurs pour chaque indice du tableau. On peut dès lors itérer via une boucle "for of" (pour les navigateurs qui la supportent) ou via la méthode `next()`.
- La méthode `Array.from()` permet de créer une nouvelle instance d'Array à partir d'un objet itérable ou semblable à un tableau.
- La méthode `map()` crée un nouveau tableau composé des images des éléments d'un tableau par une fonction donnée en argument.

Pour approfondir la question des itérateurs :

<https://x-team.com/blog/javascript-es5-array-iteration-methods-explained/>

<https://colintoh.com/blog/5-array-methods-that-you-should-use-today>

<http://www.dyn-web.com/javascript/arrays/iterate.php>

<http://2ality.com/2011/04/iterating-over-arrays-and-objects-in.html>

<https://gist.github.com/ljharb/58faf1cfcb4e6808f74aae4ef7944cff#gistcomment-1953340>

3.7 Boucles

3.7.1 Les boucles while et do

La boucle "while"

La boucle "while" est d'un usage très simple :

```
var numArray = [1,4,66,123,240,444,555];
var i = 0;
while (numArray[i] < 100) {
    console.log(numArray[i++]);
}
```

Autre exemple :

```
var numArray = [1,4,66,123,240,444,555];
var i = 0;
var len = numArray.length;
while (i < len) {
    console.log(numArray[i]);
    i++;
}
```

Dernier exemple avec lecture de la boucle "en arrière" :

```
var numArray = [1,4,66,123,240,444,555];
var i = numArray.length;
while (i--) {
    console.log(numArray[i]);
}
```

La boucle "do...while"

Si on souhaite que la boucle itère au moins une fois, et ce quelle que soit la condition d'itération définie, on peut utiliser la boucle "do...while".

Exemple :

```
var strValue = 0 ;
var nValue = 0 ;
do {
    strValue += nValue;
    nValue++;
} while (nValue <= 10) ;
console.log(strValue) ; // 55
```

3.7.2 La boucle for

La boucle "for" consiste en 3 opérations distinctes qui sont :

- une phase d'initialisation de valeur (en générale un compteur, ainsi que d'autres valeurs si nécessaire)
- une phase de test, pour vérifier que les conditions sont remplies pour autoriser une nouvelle itération
- une phase de modification du compteur utilisé pour la comparaison dans la phase précédente

Structure d'une boucle "for" :

```
for (initialisation ; évaluation de la condition ; mise à jour du compteur) { }
```

Ce qui donne en pratique :

```
for (var i = 0; i < 10; i++) {
    console.log(i);
}
```

L'approche courante pour "parcourir" un tableau consiste à écrire ceci :

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ] ;
for (var i=0; i < values.length; i++) {
    process(values[i]);
}
```

Mais c'est une mauvaise pratique, car le calcul de la longueur du tableau "values" va se répéter

à chaque itération de la boucle. Sur un tableau de grande taille, cela peut avoir un impact sur les performances (à noter que le problème est identique en PHP).

Une bien meilleure approche consiste à écrire ceci :

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ];
var i, len;
for (i=0, len=values.length ; i < len ; i+=1) {
    process(values[i]);
}
```

On a déclaré les variables "i" et "len" avant la boucle, ce qui améliore la lisibilité et réduit les risques d'erreur, et on a initialisé la valeur de la variable "len" dans la partie "initialisation" de la boucle "for" (donc au plus près du démarrage de la boucle, mais surtout pas dans la partie "évaluation").

On peut modifier le cycle d'exécution de la boucle "for" au moyen des instructions "continue" et/ou "break" :

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ];
var i, len;
for (i=0, len=values.length; i < len; i+=1) {
    if (i == 2 || i==3) {
        continue; // on "sauté" les postes numéro 2 et 3 du tableau
    } else {
        if (i == 6) {
            break; // on stopper le parcours de la boucle sur le poste 6
        }
    }
    process(values[i]);
}
```

Mais l'usage de "continue" et de "break" peut être source d'incompréhensions et d'erreurs. On peut préférer l'approche suivante, équivalente d'un point de vue fonctionnel à la précédente (mais moins performante si le tableau est amené à « grossir ») :

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ];
var i, len;
for (i=0, len=values.length; i < len; i+=1) {
    if (i < 6 && i!= 2 && i!=3) {
        process(values[i]);
    }
}
```

On peut optimiser l'exécution de la boucle "for", en lisant le tableau "en arrière" au moyen de la boucle suivante :

```
var values = [ 1, 2, 3, 4, 5, 6, 7 ];
var i;
for (i=values.length ; i-- ; ) {
    console.log(values[i]);
}
```

3.7.3 La boucle for in

La boucle "for-in" est conçue pour "itérer" sur les propriétés d'un objet.

Exemple :

```
var prop ;
var objet = {};
objet.prop1 = 'valeur prop1';
objet.prop2 = 'valeur prop2';
objet.prop3 = 'valeur prop3';
for (prop in objet) {
    console.log("Nom de propriété : " + prop);
    console.log("Valeur de propriété " + objet[prop]);
}
```

Votre objet peut éventuellement hériter des propriétés d'un autre objet, ou d'un prototype.

Si vous souhaitez ne voir que les propriétés appartenant à l'objet (et non pas celles héritées), vous pouvez utiliser la méthode `hasOwnProperty()`.

Exemple :

```
var prop ;
var objet = {};
objet.prop1 = 'valeur prop1';
objet.prop2 = 'valeur prop2';
objet.prop3 = 'valeur prop3';
for (prop in objet) {
    if (object.hasOwnProperty(prop)) {
        console.log("Nom de propriété : " + prop);
        console.log("Valeur de propriété " + objet[prop]);
    }
}
```

Attention : il ne faut pas utiliser la boucle "for-in" pour itérer sur les postes d'un tableau. Elle

n'est pas conçue pour cet usage.

Exemple de mauvaise pratique à éviter :

```
var values = [ 1, 2, 3, 4, 5, 6, 7], i;  
for (i in values) {  
    process(items[i]);  
}
```

3.8 Fonctions

3.8.1 Introduction

Les fonctions Javascript fournissent un moyen simple d'encapsuler des blocs de code réutilisables, et ainsi d'éviter autant que possible la redondance de code.

La création d'une fonction se fait de deux manières différentes, qui sont équivalentes :

```
// méthode 1 : le nom de la fonction est indiqué après l'ordre "function"  
function maFonction (arg1, arg2, ..., argn) {  
    // placer ici le code de la fonction  
    return  
}
```

Une fonction créée "à vide" n'est pas si vide que cela.

Si on utilise la fonction console.dir() sur Google Chrome, avec une fonction nouvellement créée, on constate que la fonction que l'on croyait vide, a en réalité hérité de plusieurs méthodes fournies par le constructeur Function() :

```
> var maFonction = function() {} ;
> console.dir(maFonction);
<function () {}>
  arguments: null
  caller: null
  length: 0
  name: ""
  <prototype>: Object
    <constructor>: function () {}
    <__proto__>: Object
  <__proto__>: function Empty() {}
    <apply>: function apply() { [native code] }
      arguments: null
    <bind>: function bind() { [native code] }
    <call>: function call() { [native code] }
      caller: null
    <constructor>: function Function() { [native code] }
      length: 0
      name: "Empty"
    <toString>: function toString() { [native code] }
    <__proto__>: Object
    <>Function scope</>
  <>Function scope</>
    <With Block>: CommandLineAPI
    <Global>: Window
```

Javascript offre la possibilité de déclarer des fonctions à l'intérieur d'autres fonctions, de la façon suivante :

```
function doSomethingWithItems(items) {
  var i, len, value = 10, result = value + 10;
  function doSomething(item) {
    // do something
  }
  for (i = 0, len = items.length; i < len; i++) {
    doSomething(items[i]);
  }
}
```

Dans l'exemple ci-dessus, la fonction `doSomething()` n'est utilisable à l'extérieur pas utilisable à l'extérieur de la fonction `doSomethingWithItems()`, car elle dépend pour fonctionner de paramètres qui lui sont transmis à l'intérieur de la fonction "parente". On se rapproche quelque peu de la notion de "méthode privée" telle qu'on a l'habitude de la pratiquer en PHP par exemple.

ATTENTION : la déclaration conditionnelle de fonction est en théorie autorisée, mais elle peut entraîner des comportements aléatoires sur certains navigateurs, et il est préférable de ne jamais l'employer :

```
// Bad
if (condition) {
    function doSomething() {
        console.log("Salut!");
    }
} else {
    function doSomething() {
        console.log("Yo!");
    }
}
```

On verra dans le chapitre "Fonction auto-définissables" qu'il est possible de faire quelque chose d'équivalent, mais de manière plus robuste et plus maintenable.

Pour une présentation détaillée concernant le passage de paramètres aux fonctions :

<https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Fonctions>

3.8.2 Fonctions anonymes

JavaScript permet de déclarer des fonctions anonymes. Ce sont des fonctions qui n'ont pas de noms, et qui peuvent être assignées à des variables ou à des propriétés de classes.

Exemple :

```
/*
 * méthode traditionnelle : la fonction est "complètement" créée dès le parsing du
code par l'interpréteur Javascript
 */
function doSomething() {
    // function body
};

/*
 * fonction anonyme : la variable doSomething est créée dès le parsing du code, mais
elle ne devient réellement
 * une fonction qu'au moment où l'interpréteur Javascript "passe" en exécution sur le
code de la fonction anonyme

```

```
/*
var doSomething = function() {
    // function body
};
```

Un point essentiel à comprendre avec les fonctions anonymes, c'est que le code qui se trouve à l'intérieur des accolades ne sera affecté à la variable doSomething qu'au moment où le bloc de code sera exécuté. Tant que ce code n'est pas exécuté, la variable "réceptrice" de la fonction ne contient rien d'autre que "undefined".

ATTENTION, mauvaise pratique !

On pourrait être tenté de créer les fonctions en utilisant l'opérateur "new", comme dans l'exemple suivant :

```
var maFonction = new Function (arg1, arg2, ..., argn, functionbody);
```

Cependant, l'utilisation du mot clé "new" a pour effet de bord que le code de la fonction est parsé à chaque appel, alors que ce n'est pas le cas lorsque la fonction est créée sans l'opérateur "new" (dans ce cas, le code est parsé une seule fois).

3.8.3 Fonction en exécution immédiate

Les fonctions anonymes fonctionnent en mode "exécution immédiate", si on ajoute à la fin de leur code, juste après la dernière accolade fermante, les parenthèses ouvrantes et fermantes comme dans l'exemple suivant :

```
// Bad
var value = function() {
    // function body
    return {
        message : "Salut"
    }
}();
```

Le problème avec l'exemple qui précède, c'est qu'il ressemble beaucoup à l'attribution d'une fonction anonyme à une variable. Vous n'êtes en mesure de déterminer le mode d'exécution de la fonction qu'en arrivant sur la toute dernière ligne, là où se trouvent les parenthèses ouvrantes et fermantes. Si ce détail vous a échappé, vous risquez de faire une erreur dans votre analyse du code. Cette subtilité syntaxique, un peu confuse, peut entraîner des erreurs d'interprétation de la part des développeurs. Pour y remédier, la plupart des experts Javascript

recommandent d'utiliser la syntaxe ci-dessous, qui est une très légère variante de la précédente:

```
// Good
var value = (function() {
    // function body
    return {
        message : "Salut"
    }
})();
```

Pour rendre évident le fait que la fonction s'exécute immédiatement, on l'a encapsulée dans 2 parenthèses, indiquées en rouge pour la lisibilité de l'exemple. La présence de la première parenthèse sur la première ligne est un signal fort pour le développeur qui sait dès lors à quel type de fonction il a affaire. C'est une convention, et non une obligation, mais c'est une convention vivement recommandée par la plupart des experts Javascript.

On notera qu'il n'est pas indispensable que la fonction en exécution immédiate renvoie une valeur en sortie, donc la syntaxe ci-dessous est tout aussi correcte que la précédente :

```
(function() {
    // function body
    return {
        message : "Salut"
    }
})();
```

Quel est l'intérêt des fonctions à exécution immédiate ?

Cette technique est intéressante pour toutes les opérations qui nécessitent d'être exécutées une seule fois et n'ont plus à être répétées par la suite. Par exemple, la création d'écouteurs d'évènements sur différentes éléments d'une page, est une opération qui ne doit être réalisée qu'une seule fois. On n'a ensuite plus besoin du code correspondant, il peut donc être "purgé" de la mémoire du navigateur. En encapsulant ce code dans une fonction anonyme à exécution immédiate, on est certain que le code ne sera exécuté qu'une seule fois, avant d'être purgé de la mémoire. Cet aspect est à prendre en considération pour le développement d'applications mobiles destinées à des navigateurs aux ressources (mémoire notamment) limitées.

Un autre aspect important de ce mode d'exécution, c'est qu'il constitue naturellement un mode "bac à sable" (sandbox), en évitant de polluer l'environnement avec des variables ou des fonctions, créées temporairement et qui n'ont plus d'utilité par la suite. Cela peut faciliter une cohabitation "pacifique" entre le code personnel, et le code de libraries ou frameworks utilisés conjointement.

Peut-on passer des paramètres à une fonction à exécution immédiate ?

Oui, en transmettant les valeurs dans le dernier groupe de parenthèses (cf. les variables valprm1 et valprm2) :

```
var value = (function(param1, param2) {
    // function body
    return {
        message : "Salut " + param1 + param2 ;
    }
})(valprm1, valprm2));
```

A noter : Les anglophones désignent cette technique sous le terme suivant : "immediately-invoked function expression" (abrégé en "IIFE").

3.8.4 Portée des variables (scope)

La notion de portée (en anglais : "scope") est fondamentale en Javascript, plus encore que dans tout autre langage.

Une mauvaise compréhension de cette notion peut entraîner des problèmes en cascade.

Une variable déclarée à l'intérieur d'une fonction avec le mot clé "var" est une variable "locale" à cette fonction. On dit donc que sa portée est "locale", elle n'est pas accessible en dehors du corps de la fonction.

Une variable qui est utilisée à l'intérieur d'une fonction sans avoir été déclarée à l'intérieur de cette même fonction est considérée comme "globale".

Pour bien comprendre, prenons l'exemple d'une variable "myglobal", utilisée à l'intérieur d'une fonction, sans y avoir été déclarée au préalable au moyen du mot clé "var".

```
function test() {
    myglobal = "hello"; // mauvaise pratique (antipattern)
    console.log('1=' + myglobal); // "1=hello"
    console.log('2=' + window.myglobal); // "2=hello"
    console.log('3=' + window["myglobal"]); // "3=hello"
    console.log('4=' + this.myglobal); // "4=hello"
}
test();
```

On le voit dans l'exemple ci-dessus, le fait d'avoir utilisé la variable "myglobal" sans l'avoir déclarée localement, a pour effet que cette variable se retrouve automatiquement attachée à

l'objet "window". L'environnement de l'objet "window" est du coup "pollué" par une donnée qui ne lui était pas destinée.

On voit également, que le mot clé "this" (dont on reparlera plus en détail dans le chapitre consacré aux objets), utilisé à l'intérieur d'une fonction qui n'est pas instanciée sous forme d'un objet, fait lui aussi référence à l'objet "window".

Pour éviter ce genre de problème, on recommandera d'utiliser systématiquement l'instruction "use strict" qui sera présentée plus en détails au chapitre suivant.

```
function test() {  
    "use strict";  
    myglobal = "hello"; // mauvaise pratique (antipattern)  
    console.log('1=' + myglobal); // "hello"  
    console.log('2=' + window.myglobal); // "hello"  
    console.log('3=' + window["myglobal"]); // "hello"  
    console.log('4=' + this.myglobal); // "hello"  
}  
test(); // TypeError: this is undefined
```

On voit que Javascript a changé de comportement, avec la présence de la directive "use strict". L'exécution de la fonction n'est plus possible, compte tenu des erreurs qu'elle contient.

3.8.5 Mode Strict

La norme ECMAScript 5 (ES5) a introduit le mode "strict". Ce mode permet d'influer sur la manière dont Javascript parse et exécute le code. L'objectif est de réduire au maximum les erreurs de syntaxe, et donc d'améliorer la robustesse générale du code.

Pour mettre un script en mode "strict", il faut utiliser la syntaxe suivante :

```
"use strict";
```

Cela ressemble vaguement à une chaîne de caractères qui ne serait pas affectée à une variable, mais en réalité le moteur Javascript pour ES5 traite cette ligne comme une commande qui lui permet de basculer en mode "strict".

Cette commande est valide aussi bien globalement (à l'extérieur de toute fonction) que localement (à l'intérieur des fonctions). Cependant, il est vivement recommandé de ne l'utiliser qu'à l'intérieur des fonctions, donc sur une portée "locale" (en anglais "local scope"). La raison en est que le mode strict appliqué "globalement" s'applique sur l'ensemble du code Javascript.

Si vous utilisez du code Javascript de diverses origines (par exemple : un framework, quelques plugins, quelques fonctions "maison", etc...), le risque est grand que certaines portions de code ne soient pas compatibles avec le mode "strict". Cela pourrait entraîner des erreurs bloquant l'exécution du code Javascript dans son ensemble. Pour cette raison, il est préférable de ne jamais utiliser le mode "strict" avec une portée globale.

Voici quelques exemples :

```
// Mauvais, car le mode strict s'applique globalement (risque d'effets de bord)
"use strict";
function doSomething() {
    // placer votre code ici
}

// Très bon :)
function doSomething() {
    "use strict";
    // placer votre code ici
}
```

Si vous souhaitez appliquer le mode "strict" à plusieurs fonctions, sans vous retrouver avec la corvée consistant à insérer la commande au début de chaque fonction, vous pouvez recourir à la technique de l'invocation immédiate de code, comme dans l'exemple suivant :

```
// Good
(function() {
    "use strict";
    function doSomething() {
        // code
    }
    function doSomethingElse() {
        // code
    }
})();
```

Dans cet exemple, les fonctions doSomething() et doSomethingElse() s'exécutent toutes deux en mode "strict", car elles sont contenues dans une fonction anonyme invoquée immédiatement.

L'utilisation du mode "strict" est vivement recommandée, à condition de respecter les conditions indiqués ci-dessus.

Pour bien comprendre l'enjeu, il faut souligner que le cœur du langage Javascript est basé sur la norme ECMAScript. La version 3 est "sortie" officiellement en 1999, et elle est bien supportée par la plupart des navigateurs. Si la version 4 fut abandonnée, la version 5 est sortie en décembre 2009. Cette version 5 apporte de nouveaux objets prédéfinis, de nouvelles méthodes et propriétés, et surtout le support du mode strict.

Une évolution annoncée du langage serait que à moyen terme, le mode strict devienne le mode de fonctionnement par défaut du langage, et peut être même le seul mode disponible.

3.8.6 Namespaces

La notion d'espace de nommage (ou "Namespace") que l'on rencontre dans d'autres langages, n'existe pas en Javascript (en tout cas pas sur ES5). Mais on peut contourner le problème très facilement.

Si l'on reprend la fonction MyEventManager étudiée dans l'introduction du chapitre "DOM Events", on peut l'encapsuler dans un objet plus important, appelé MyApp, comme dans l'exemple suivant :

```
MyApp = {};
MyApp.MyEventManager = {
    addListener : function(eventObj, eventName, eventHandler) {
        if (eventObj.addEventListener) {
            // solution compatible tous navigateurs sauf IE < 9
            return eventObj.addEventListener(eventName, eventHandler, false);
        } else if (eventObj.attachEvent) {
            // pour IE < 9
            return eventObj.attachEvent("on" + eventName, eventHandler);
        } else {
            // solution de la dernière chance
            return eventObj["on" + eventName] = eventHandler;
        }
    },
    removeListener : function(eventObj, eventName, eventHandler) {
        if (eventObj.removeEventListener) {
            eventObj.removeEventListener(eventName, eventHandler, false);
        } else if (eventObj.detachEvent) {
            eventObj.detachEvent("on" + eventName, eventHandler);
        } else {
            eventObj["on" + eventName] = null;
        }
    }
}
```

A l'usage, on pourra faire appel à la méthode addListener() qui est "attachée" à la méthode MyEventManager(), elle-même "attachée" à l'objet MyApp.

```
MyApp.MyEventManager.addListener(...);
```

Quelques conseils :

L'écriture de l'objet MyApp peut vite devenir fastidieuse, si on encapsule toutes les méthodes selon le modèle ci-dessus.

Pour bénéficier de plus de souplesse, on pourra utiliser la variante suivante :

```
MyApp = {};
MyApp.MyEventManager = {} ;
MyApp.MyEventManager.addEventListener = function(eventObj, eventName, eventHandler) {
    if (eventObj.addEventListener) {
        // solution compatible tous navigateurs sauf IE < 9
        return eventObj.addEventListener(eventName, eventHandler, false);
    } else if (eventObj.attachEvent) {
        // pour IE < 9
        return eventObj.attachEvent("on" + eventName, eventHandler);
    } else {
        // solution de la dernière chance
        return eventObj["on" + eventName] = eventHandler;
    }
};

MyApp.MyEventManager.removeListener = function(eventObj, eventName, eventHandler) {
    if (eventObj.removeEventListener) {
        eventObj.removeEventListener(eventName, eventHandler, false);
    } else if (eventObj.detachEvent) {
        eventObj.detachEvent("on" + eventName, eventHandler);
    } else {
        eventObj["on" + eventName] = null;
    }
};
```

De plus, si à l'intérieur de l'objet MyApp.MyEventManager, on souhaite que plusieurs méthodes fassent appel les unes aux autres, il peut être intéressant de disposer d'un mot clé "self", faisant référence à l'objet MyApp, de façon à ne pas répéter ce nom explicitement. Si par la suite on souhaite remplacer "MyApp" par un autre nom, il ne sera pas nécessaire de remplacer "MyApp" partout dans le code.

```
MyApp = {};
MyApp.self = MyApp ; // ligne ajoutée pour disposer d'un pointeur "self" sur MyApp
MyApp.MyEventManager = {} ;
...
...
```

3.8.7 Fonctions en rappel (callback)

Les fonctions étant des objets, elles peuvent être passées en paramètres à d'autres fonctions. Les anglophones désignent cette technique sous le terme de "callback".

Exemple :

```
function maFonction1(callback) {
    console.log('un petit coucou dans maFonction1 avant de passer la main à la
fonction "callback"');

    /* il est toujours bon de vérifier que callback est défini et qu'il s'agit
bien d'une fonction */
    if (callback && typeof callback === 'function') {
        callback();
    } else {
        console.log('callback n\'est pas définie ou n\'est pas une fonction') ;
    }
}

function maFonction2() {
    console.log('on se trouve dans maFonction2');
}

maFonction1(maFonction2); // attention à ne pas mettre de parenthèses () derrière
maFonction2
```

Ce principe est abondamment utilisé en Javascript, c'est typiquement celui que l'on utilise quand on attache un écouteur d'évènement de type "onclick" sur un bouton par exemple :

```
var doSomething = function() {
    console.log('ça marche !');
}

var btn = document.getElementById("action-btn");
btn.addEventListener('click', doSomething, false);
```

Il faut faire très attention à ne pas placer de parenthèses derrière la fonction doSomething, par exemple, le code suivant est incorrect, car il suppose que la fonction doSomething() va être exécutée immédiatement, alors que ce n'est pas du tout l'objectif visé :

```
var btn = document.getElementById("action-btn");
btn.addEventListener('click', doSomething(), false);
```

Dans certains cas, la fonction callback peut être une méthode d'un objet. Dans ce cas de figure, le plus simple est de transmettre à la fonction appelante deux paramètres au lieu d'un :

- 1er paramètre : la méthode "callback",
- 2ème paramètre : l'objet auquel la méthode est attachée

Exemple :

```
var fonctionAppelante = function(callback, callback_objet) {
    //...
    if (typeof callback === "function") {
        callback.call(callback_objet, found);
    }
    // ...
};
```

On peut aussi préférer que le paramètre "callback" soit transmis sous la forme d'une chaîne (String), dans ce cas on peut écrire :

```
var fonctionAppelante = function(callback, callback_objet) {
    //...
    if (typeof callback === "string") {
        callback = callback_objet[callback];
    }
    // ...
};
```

3.8.8 Fonctions auto-définissables

Les fonctions Javascript ont la possibilité de redéfinir leur propre code dynamiquement. On dit qu'elles sont auto-définissables (en anglais : "Self-Defining Functions").

Exemple :

```
var faisMoiPeur = function() {
    console.log("Bouh!");
    faisMoiPeur = function() {
        console.log("Bouh! Bouh!");
    };
}
// Premier appel de la fonction "auto-définissable"
faisMoiPeur(); // Bouh!
// Second appel où l'on voit que le code de la fonction faisMoiPeur() a changé
faisMoiPeur(); // Bouh! Bouh!
```

Dans quel cas cette fonctionnalité pourrait-elle être utile ?

Un exemple d'utilisation possible est le cas où l'on doit vérifier si le navigateur supporte une fonctionnalité.

Prenons l'exemple de la fonction addListener() présentée plus loin, dans l'introduction du chapitre "Dom Events" :

```
function addListener(eventObj, eventName, eventHandler) {
    if (eventObj.addEventListener) {
        // solution compatible tous navigateurs sauf IE < 9
        return eventObj.addEventListener(eventName, eventHandler, false);
    } else if (eventObj.attachEvent) {
        // pour IE < 9
        return eventObj.attachEvent("on" + eventName, eventHandler);
    } else {
        // solution de la dernière chance
        return eventObj["on" + eventName] = eventHandler;
    }
}
```

Si cette fonction est exécutée plusieurs fois - et ce sera très certainement le cas - elle va effectuer systématiquement les mêmes contrôles consistant à vérifier si le navigateur supporte les méthodes addEventListener(), attachEvent(), etc...

En reprenant le principe présenté dans la fonction faisMoiPeur(), on peut réécrire la fonction addListerer() de la façon suivante :

```
var addListener = function(eventObj, eventName, eventHandler) {
    var tmpEvent = null;
    if (eventObj.addEventListener) {
        // solution compatible tous navigateurs sauf IE < 9
        function addListener(eventObj, eventName, eventHandler) {
            return eventObj.addEventListener(eventName, eventHandler, false);
        }
    } else if (eventObj.attachEvent) {
        // pour IE < 9
        function addListener(eventObj, eventName, eventHandler) {
            tmpEvent = eventObj.attachEvent("on" + eventName, eventHandler);
        }
    } else {
        // solution de la dernière chance
        function addListener(eventObj, eventName, eventHandler) {
            tmpEvent = eventObj["on" + eventName] = eventHandler;
        }
    }
    return addListener(eventObj, eventName, eventHandler);
};
```

Au premier appel, la fonction va exécuter le code ci-dessus, mais à la seconde exécution, en

supposant que le navigateur supporte la méthode addEventListener(), le code exécuté par la même fonction sera le suivant :

```
function addListener(eventObj, eventName, eventHandler) {  
    return eventObj.addEventListener(eventName, eventHandler, false);  
}
```

3.8.9 Crédit à la fonction fléchée sous ES6 (« arrow function »)

L'exemple de code suivant, écrit selon la syntaxe ES6 :

```
let add = (x,y) => x + y;  
  
let result = add(1,1);  
  
console.log(result);
```

... peut être converti de la façon suivante sous ES5 :

```
"use strict";  
  
var add = function (x, y) {  
    return x + y;  
};  
  
var result = add(1, 2);  
  
console.log(result);
```

Les fonctions fléchées ne permettent pas l'utilisation de « this ». Elles ne peuvent dès lors pas être utilisées comme bases pour l'instanciation d'objets.

3.9 *Objets*

3.9.1 Introduction

Le langage ECMAScript (Javascript pour le grand public) est un langage un peu déroutant, au premier abord, pour les développeurs habitués au modèle objet d'autres langages, comme Java ou PHP. Si en Java et en PHP, un objet est instancié à partir d'une classe, on verra dans un instant que ce n'est pas du tout le cas en Javascript.

Quand on commence à développer en Javascript, on s'aperçoit que très peu d'éléments échappent à la notion d'objets. En Javascript, les fonctions sont des objets, elles peuvent avoir des méthodes et des propriétés.

Cinq variables primitives ne sont pas des objets, ce sont les variables de type : number, string, boolean, null, et undefined.

Les variables de type number, string et boolean peuvent être associés à des objets wrapper de même nom, pour être manipulés comme des objets. Cette mécanique peut être mise en oeuvre par le développeur, mais le plus souvent elle est déclenchée automatiquement par le moteur Javascript lui-même (quand par exemple on utilise une méthode "length" sur une variable de type "string").

3.9.2 Objet personnalisé

En Javascript, un objet est un container pour une collection de paire "clés-valeurs". Ces paires de "clés-valeurs" peuvent correspondre à des variables littérales ou à des objets, on dit dans ce cas qu'il s'agit de propriétés. Elles peuvent aussi correspondre à des appels de fonctions, on dit dans ce cas qu'il s'agit de méthodes.

Pour créer un objet, on peut utiliser les 2 syntaxes suivantes, qui sont strictement équivalentes :

```
var monObjet = {} ;  
var monObjet = new Object() ;
```

Un objet créé de cette façon n'a encore ni propriété, ni méthode, en propre, mais il n'est pas complètement vide puisqu'il hérite d'un certain nombre de méthodes fournies par le constructeur Object(). On le voit très clairement si on applique la fonction console.dir() à l'objet monObjet créé précédemment :

```

__proto__: Object
__defineGetter__: function __defineGetter__() { [native code] }
__defineSetter__: function __defineSetter__() { [native code] }
__lookupGetter__: function __lookupGetter__() { [native code] }
__lookupSetter__: function __lookupSetter__() { [native code] }
constructor: function Object() { [native code] }
hasOwnProperty: function hasOwnProperty() { [native code] }
isPrototypeOf: function isPrototypeOf() { [native code] }
propertyIsEnumerable: function propertyIsEnumerable() { [native code] }
toLocaleString: function toLocaleString() { [native code] }
toString: function toString() { [native code] }
valueOf: function valueOf() { [native code] }
get __proto__: function __proto__() { [native code] }
set __proto__: function __proto__() { [native code] }

```

Attention : l'attribut `__proto__` n'existe pas en tant que tel dans la norme Javascript. C'est simplement une facilité offerte par Firebug et Google Chrome pour accéder, à partir d'un objet, aux propriétés et méthodes de son objet parent (dont elle hérite au travers de la propriété "prototype").

Les objets en Javascript sont très souples d'utilisation. On peut les enrichir en leur affectant des méthodes et propriétés, au fur et à mesure des besoins :

```

var monObjet = {} ;
monObjet.projet = 'Cours Javascript' ;
monObjet.duree = 4 ;
monObjet.statut = 'ready' ;
monObjet.getPrice = function () {
    var prix = 0 ;
    // implémenter un algorithme de calcul de prix ici
    return prix ;
} ;
console.log(monObjet) ; // Object { projet="Cours Javascript", duree=4,
statut="ready", more...}

```

On peut créer le même objet `monObjet` en utilisant une autre technique, qui consiste à créer tout d'abord une fonction, fonction que nous allons utiliser ensuite comme "constructeur" pour instancier 3 objets différents.

```

var MonConstructeur = function (projet, duree, statut) {
    this.projet = projet ;
    this.duree = duree ;
    this.statut = statut ;
    this.getPrice = function () {
        var prix = 0 ;
        // implémenter un algorithme de calcul de prix ici
        return prix ;
    }
}

```

```
    } ;
}
var monProjet1 = new MonConstructeur ('Cours Javascript', 4, 'ready') ;
console.log(monProjet1); // Object { projet="Cours Javascript", duree=4,
statut="ready", more...}

var monProjet2 = new MonConstructeur ('Cours PHP', 4, 'ready') ;
console.log(monProjet2); // Object { projet="Cours PHP", duree=4, statut="ready",
more...}

var monProjet3 = new MonConstructeur ('Cours SQL DB2', 4, 'ready') ;
console.log(monProjet3); // Object { projet="Cours SQL DB2", duree=4,
statut="ready", more...}
```

Explication : une fonction "constructeur" permet de produire plusieurs objets à partir d'une même structure. Les objets obtenus ont les mêmes caractéristiques que l'objet "monObjet" créé précédemment, ils offrent la même souplesse et notamment la possibilité d'enrichir la structure de l'un ou l'autre des objets monProjet1, monProjet2 ou monProjet3.

Points importants :

- une convention largement admise par les développeurs Javascript consiste à mettre en majuscule le premier caractère du nom de la fonction constructeur. Cette convention vise à distinguer une fonction qui sera utilisée uniquement comme fonction, d'une fonction qui sera utilisée comme constructeur d'objet via le mot clé "new"
- il faut être vigilant sur le fait que le mot clé "this" utilisé dans la fonction constructeur appartient à l'objet instancié, à la condition que l'objet instancié le soit via le mot clé "new". Si l'objet n'est pas instancié avec le mot clé "new", alors "this" correspond à l'objet "parent" de l'objet courant. Si on se retrouve dans cette situation, on risque de "polluer" un objet de plus haut niveau qui n'avait rien demandé.
- un objet instancié à partir d'une fonction constructeur peut avoir des propriétés et des méthodes propres, qui viennent compléter, voire remplacer, les propriétés et méthodes fournies par la fonction constructeur.

Qu'est ce que "this" ? Que se passe-t-il dans les coulisses de l'objet au moment de son instantiation ?

Le code de la fonction constructeur est le suivant :

```
var MonConstructeur = function (projet, duree, statut) {
    this.projet = projet ;
    this.duree = duree ;
    this.statut = statut ;
    this.getPrice = function () {
        var prix = 0 ;
        // implémenter un algorithme de calcul de prix ici
        return prix ;
    } ;
}
```

Mais dans les coulisses, voici ce qui se passe :

```
var MonConstructeur = function (projet, duree, statut) {
    var this = {} ; // objet this créé automatiquement par le moteur Javascript
                    // lors de l'instanciation avec "new"
    this.projet = projet ;
    this.duree = duree ;
    this.statut = statut ;
    this.getPrice = function () {
        var prix = 0 ;
        // implémenter un algorithme de calcul de prix ici
        return prix ;
    } ;
    return this ; // renvoi de this en sortie de l'objet instancié
}
```

Peut-on modifier le comportement par défaut de la fonction constructeur ?

La réponse est oui, à condition de renvoyer un objet en sortie de la fonction. Dans l'exemple ci-dessous, on a déclaré notre propre objet "that" et c'est lui que l'on renvoie en sortie de l'objet.

```
var MonConstructeur = function(projet, duree, statut) {
    var that = {}; // objet that créé explicitement par le développeur en
                   // remplacement de this
    that.projet = projet;
    that.duree = duree;
    that.statut = statut;
    that.getPrice = function() {
        var prix = 0;
        // implémenter un algorithme de calcul de prix ici
        return prix;
    };
    return that; // renvoi de that en sortie de l'objet instancié
}
```

```
var monProjet1 = new MonConstructeur('Cours Javascript', 4, 'ready');
console.log(monProjet1); // Object { projet="Cours Javascript", duree=4,
statut="ready", more...}
```

3.9.3 Prototype

La fonction constructeur présentée au chapitre précédent présente un gros défaut.

La méthode getPrice() se trouvant embarquée dans la fonction constructeur, son code sera dupliqué dans chacun des objets instanciés.

Il serait intéressant de pouvoir placer ce code à un niveau supérieur, or ce niveau supérieur existe, c'est la propriété "prototype" associée à la fonction constructeur.

En sortant la méthode getPrice() de la fonction constructeur, et en l'associant à la propriété prototype, le code devient disponible pour chaque objet instancié, sans pour autant être dupliqué dans chacun de ces objets.

```
var MonConstructeur = function (projet, duree, statut) {
    this.projet = projet ;
    this.duree = duree ;
    this.statut = statut ;
}

// Il peut être intéressant de vérifier si la méthode getPrice() n'est pas déjà
// définie dans la fonction constructeur
if (typeof MonConstructeur.prototype.getPrice === "undefined") {

    MonConstructeur.prototype.getPrice = function (taxe) {
        var prix = 5 ;
        // implémenter un algorithme de calcul de prix ici
        prix = prix + ( prix * taxe / 100 ) ;
        return prix ;
    } ;
}

var monProjet1 = new MonConstructeur ('Cours Javascript', 4, 'ready') ;
console.log(monProjet1.getPrice(4)); // 5.2

var monProjet2 = new MonConstructeur ('Cours PHP', 4, 'ready') ;
console.log(monProjet1.getPrice(5)); // 5.25

var monProjet3 = new MonConstructeur ('Cours SQL DB2', 4, 'ready') ;
console.log(monProjet1.getPrice(6)); // 5.3
```

La méthode hasOwnProperty() permet de vérifier si une propriété appartient "en propre" à l'objet considéré (et pas par héritage) :

```
console.log(monProjet1.hasOwnProperty('projet')) ; // true
console.log(monProjet1.hasOwnProperty('xrojet')) ; // false
console.log(monProjet1.hasOwnProperty('getPrice')) ; // false
```

3.9.4 Constructeurs natifs

Le langage Javascript contient 9 objets constructeurs natifs, qui sont :

- Number()
- String()
- Boolean()
- Object()
- Array()
- Function()
- Date()
- RegExp()
- Error()

Par exemple, les fonctions sont des objets créés via le constructeur Function().

Points importants :

- L'objet Math() ne fait pas partie de la liste ci-dessus, car c'est un objet statique, et non pas un objet constructeur. C'est un regroupement de méthodes et propriétés destinés aux opérations mathématiques (comme Math.PI par exemple).
- Les constructeurs Number(), String(), et Boolean() ont pour particularité de servir aussi de "wrappers" pour la manipulation des primitives string, number et boolean.

Exemples d'utilisation des 9 constructeurs natifs de Javascript :

```
// tester les 9 constructeurs natifs
var myNumber = new Number(23);
var myString = new String('male');
var myBoolean = new Boolean(false);
var myObject = new Object();
var myArray = new Array('foo','bar');
var myFunction = new Function("x", "y", "return x*y");
var myDate = new Date();
var myRegExp = new RegExp('\bt[a-z]+\b');
var myError = new Error('Crap!');

// vérifier le constructeur associé à chaque objet
console.log(myNumber.constructor); // logs Number()
console.log(myString.constructor); // logs String()
```

```
console.log(myBoolean.constructor); // logs Boolean()
console.log(myObject.constructor); // logs Object()
console.log(myArray.constructor); // logs Array(), in modern browsers
console.log(myFunction.constructor); // logs Function()
console.log(myDate.constructor); // logs Date()
console.log(myRegExp.constructor); // logs RegExp()
console.log(myError.constructor); // logs Error()
```

3.9.5 A quel objet appartient une instance

Il est souvent utile de pouvoir vérifier à quel constructeur d'objet est lié une instance. Pour cela, on utilisera le mot clé "instanceof".

Attention : instanceof ne fonctionne vraiment qu'avec des objets et des instances créées à partir de fonctions personnalisées. Dans le cas de primitives (Number, String et Boolean), et même dans le cas où on aurait utilisé les équivalents wrappers Number, String et Boolean, instanceof renverra "false" systématiquement.

Exemple :

```
// création d'une variable primitive
var maChaine = "ceci est ma chaîne";
console.log(maChaine instanceof String); // false

// un tableau n'est pas une primitive, instanceof devrait donc renvoyer true
var monTableau = ['1', '2', '3'];
console.log(monTableau instanceof Array); // true

// création d'un constructeur personnalisé
var CustomConstructor = function() {this.foo = 'bar'};

// création d'un objet à partir du constructeur "maison"
var instanceOfCustomObject = new CustomConstructor();
console.log(instanceOfCustomObject instanceof CustomConstructor); // true
```

3.9.6 Création d'objets sous ES6, avec et sans transpiler

Peut-on faire de l'héritage selon l'approche objet "traditionnelle" en Javascript ?

Oui, c'est un peu plus compliqué qu'en PHP, mais c'est possible en ES5.

Le moteur graphique pour la construction de jeu et d'animations Tomahawk propose un exemple intéressant d'implémentation de l'héritage en JS :

```
/**  
 * @namespace  
 */  
var tomahawk_ns = new Object();  
tomahawk_ns.version = "0.9.1";  
  
/**  
 * @class Tomahawk  
 * Core Framework class  
 * @constructor  
 */  
function Tomahawk() {  
}  
  
Tomahawk._classes = new Object();  
Tomahawk._extends = new Array();  
  
Tomahawk._funcTab = null;  
  
Tomahawk.registerClass = function (classDef, className){  
    Tomahawk._classes[className] = classDef;  
};  
  
Tomahawk.extend = function (p_child, p_ancestor){  
    Tomahawk._extends.push({"child": p_child, "ancestor": p_ancestor, "done":  
false});  
};  
  
Tomahawk.run = function (){  
    var obj = null;  
    var i = 0;  
    var max = Tomahawk._extends.length;  
  
    Tomahawk._funcTab = new Object();  
  
    for (i = 0; i < max; i++){  
        obj = Tomahawk._extends[i];  
        Tomahawk._inherits(obj);  
    }  
}
```

```

Tomahawk._getParentClass = function (child){
    var i = 0;
    var max = Tomahawk._extends.length;
    for (i = 0; i < max; i++){
        obj = Tomahawk._extends[i];
        if (obj["child"] == child){
            return obj;
        }
    }
    return null;
};

Tomahawk._inherits = function (obj){
    var child = null;
    var ancestor = null;
    var superParent = Tomahawk._getParentClass(obj["ancestor"]);

    if (superParent != null && superParent.done == false){
        Tomahawk._inherits(superParent);
    }
    child = Tomahawk._classes[obj["child"]];
    ancestor = Tomahawk._classes[obj["ancestor"]];
    obj.done = true;
    var obj = new Object();
    if (child != null && ancestor != null){
        for (var prop in ancestor.prototype){
            obj[prop] = ancestor.prototype[prop];
        }
        for (var prop in child.prototype) {
            obj[prop] = child.prototype[prop];
        }
    }
    child.prototype = obj;
};

```

Avec le principe appliqué dans Tomahawk, on peut écrire par exemple ceci :

```

// fonction constructeur
function Voiture(name,id){
    this.name = name;
    this.id = id;
}

Tomahawk.registerClass(Voiture, 'Voiture');

Voiture.prototype.name = null;
Voiture.prototype.id = null;
Voiture.prototype.noise = function () {
    console.log(this);
} ;

```

```
function Chevrolet(name,id){  
    Voiture.apply(this, [name, id]) ;  
}  
Tomahawk.registerClass(Chevrolet, 'Chevrolet');  
Tomahawk.extend('Chevrolet', 'Voiture');  
  
Chevrolet.prototype.noise = function () {  
    Voiture.prototype.noise.apply(this);  
    console.log('vroum');  
} ;  
  
function Impala67 (name, id) {  
    Chevrolet.apply(this, [name, id]);  
}  
Tomahawk.registerClass(Impala67, 'Impala67');  
Tomahawk.extend('Impala67', 'Chevrolet');  
  
Impala67.prototype.noise = function () {  
    Chevrolet.prototype.noise.apply(this) ;  
    console.log('Yes!!!!') ;  
} ;  
  
Tomahawk.run();  
var chevrolet = new Chevrolet("chevrolet",1);  
var clio = new Voiture("clio",2);  
chevrolet.noise() ;  
clio.noise() ;  
var impala67 = new Impala67 ('supernatural' , 3);  
impala67.noise() ;
```

Site officiel de Tomahawk :

<http://the-tiny-spark.com/tomahawk/>

Avec l'arrivée de la norme EcmaScript 6 (version 2015), il devient possible d'implémenter l'héritage plus simplement.

Dans son livre “Exploring ES6”, Axel Rauschmayer (cf. bibliographie) propose un exemple d'implémentation en ES5, et son équivalent avec ES6, d'une classe “fille” *Employee* qui est dérivée d'une classe “mère” *Person* :

- Classe “mère” *Person* et classe “fille” *Employee* en version ES5 :

```
function Person(name) {
    this.name = name;
}
Person.prototype.describe = function () {
    return 'Person called ' + this.name;
};

function Employee(name, title) {
    Person.call(this, name); // super(name)
    this.title = title;
}
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;

Employee.prototype.describe = function () {
    return Person.prototype.describe.call(this) // super.describe()
        + ' (' + this.title + ')';
};
```

- Classe “mère” *Person* et classe “fille” *Employee* en version ES6 :

```
class Person {
    constructor(name) {
        this.name = name;
    }
    describe() {
        return 'Person called ' + this.name;
    }
}

class Employee extends Person {
    constructor(name, title) {
        super(name); this.title = title;
    }
    describe() {
        return super.describe() + ' (' + this.title + ')';
    }
}
```

Le projet BabelJS est un “transpileur”, c'est à dire un traducteur dont le rôle consiste à transformer du code ES6 en code ES5, de manière à garantir le bon fonctionnement d'une application sur la grande majorité des navigateurs. Par exemple, le code ci-dessous :

```
class Employee extends Person {
    constructor(name, title) {
        super(name); this.title = title;
    }
    describe() {
        return super.describe() + ' (' + this.title + ')';
    }
}
```

... traduit en code ES6 par BabelJS devient :

```
'use strict';

var Employee = function (_Person) {
    _inherits(Employee, _Person);

    function Employee(name, title) {
        _classCallCheck(this, Employee);

        var _this = _possibleConstructorReturn(this,
            (Employee.__proto__ || Object.getPrototypeOf(Employee)).call(this, name));

        _this.title = title;
        return _this;
    }

    _createClass(Employee, [
        {
            key: 'describe',
            value: function describe() {
                return _get(Employee.prototype.__proto__ ||
                    Object.getPrototypeOf(Employee.prototype), 'describe', this).call(this) +
                    ' (' + this.title + ')';
            }
        }]);
}

return Employee;
}(Person);
```

On notera que BabelJS fournit un jeu de fonctions spécifiques (`_inherits`, `_classCallCheck`, `_createClass`, `_get`) qui facilitent le portage du code ES6 vers ES5. Ces fonctions n'ont pas été intégrées au code de l'exemple précédent afin de ne pas l'alourdir.

Le site officiel de BabelJS est le suivant :

<https://babeljs.io/repl/>

Article intéressant sur BabelJS :

<http://www.thegeekstuff.com/2016/12/babel-for-javascript/>

BabelJS n'est pas le seul trancpiler disponible. Google propose Traceur, qui fournit à peu près les mêmes services :

<http://google.github.io/traceur-compiler/demo/repl.html#>

3.9.7 Sécuriser les objets

Avec ES5 est apparue la possibilité de sécuriser les objets Javascript, avec 3 niveaux de protection différents :

- Prevent extension : Aucune nouvelle propriété ou méthode ne peut être ajoutée à l'objet, mais celle-ci peut être modifiée ou supprimée.
- Seal (en anglais = « scellé ») : reprend les règles de « Prevent extension » en y ajoutant l'impossibilité de supprimer les propriétés et méthodes existantes
- Freeze : Identique à "Seal", mais toute modification des propriétés et méthodes est impossible (tous les champs sont en lecture seule)..

Pour activer le mode « Prevent extension » sur un objet, il faut utiliser la méthode Object.preventExtension(), et pour savoir si un objet est verrouillé selon cette méthode, il faut utiliser la méthode Object.isExtensible(person).

```
var person = { name: "Nicholas" };

// lock down the object
Object.preventExtensions(person);

console.log(Object.isExtensible(person)); // false
```

Pour sceller un objet (Seal), il faut utiliser la méthode Object.seal(). On peut déterminer si un objet est déjà « scellé » en utilisant la méthode Object.isSealed().

Pour mettre un objet à l'état "freeze", il faut utiliser Object.freeze(). On peut savoir si un objet est gelé en utilisant la méthode Object.isFrozen().

On ne peut pas dire que les techniques présentées dans ce chapitre aient connu un grand succès auprès des développeurs, peut être du fait qu'il est difficile de trouver des polyfills

satisfaisants pour les plus vieux navigateurs. Nous verrons au chapitre 3.10.2 une méthode permettant de définir une notion de visibilité (publique/privée) avec une bonne compatibilité « cross browser ».

3.9.8 Conclusion

Le présent chapitre avait pour objectif de présenter les bases du modèle objet de Javascript, de manière à être en mesure d'utiliser ce modèle pour les opérations de manipulation du DOM qui seront abordées dans les chapitres suivants.

On l'aura compris, le modèle objet de Javascript, fondé sur la notion de prototype, n'est pas un modèle objet "au rabais". Au contraire, c'est un modèle très souple et en définitive très puissant.

Pour un approfondissement sur ce sujet, on recommandera la lecture des livres suivants, tous édités chez O'Reilly :

- "Javascript Patterns", de Stoyan Stefanov
- "Learning the Javascript Design Patterns", de Addy Osmani
- "You don't know JS : this and Object Prototypes", de Kyle Simpson
- "You don't know JS : Scope and Closures", de Kyle Simpson

Nous avons également évoqué certaines caractéristiques de la norme ES6 en matière de définition des objets. Pour un tour d'horizon plus complet de ES6, voir le chapitre "bibliographie".

Dans le chapitre suivante, nous allons étudier un certain nombre de bonnes pratiques permettant d'améliorer la robustesse et la maintenabilité du code.

3.10 Bonnes pratiques POO

3.10.1 Généralités

On a vu que les constructeurs sont de simples fonctions, et que les objets sont instanciés avec "new".

Que se passe-t-il si on oublie d'utiliser "new" en invoquant une fonction constructeur ? Il n'y aura pas d'anomalie, mais l'objet "this" utilisé à l'intérieur de la fonction constructeur fera référence à un objet "parent" (dans les navigateurs ce sera généralement l'objet window), au lieu de faire référence à l'objet courant.

```
// constructeur
function Gaufre() {
    this.tastes = "délicieux";
}
// instantiation d'un objet dans les règles de l'art
var good_morning = new Gaufre();
console.log(typeof good_morning); // "object"
console.log(good_morning.tastes); // "délicieux"
console.log(good_morning.constructor); // Gaufre()

// antipattern: new a été oublié
var good_morning = Gaufre();
console.log(typeof good_morning); // "undefined"
console.log(window.tastes); // "délicieux"
console.log(good_morning.constructor); // TypeError : good_morning is undefined
```

Si on utilise le mode strict de la norme ECMAScript 5, alors le comportement change :

```
// constructeur
function Gaufre() {
    "use strict";
    this.tastes = "délicieux";
}

// le mode strict ne permet pas d'utiliser "this" sur un constructeur non instancié
var good_morning = Gaufre(); // TypeError : this is undefined
```

Constructeur auto-invocable

On peut prévenir tout risque d'oubli du mot clé "new" en ajoutant quelques lignes à l'intérieur de la fonction constructeur Gaufre() :

```
// constructeur
function Gaufre() {
    if (!(this instanceof Gaufre)) {
        return new Gaufre();
    }
    this.tastes = "délicieux";
}
// instantiation d'un objet dans les règles de l'art
var good_morning = new Gaufre();
console.log(typeof good_morning); // "object"
console.log(good_morning.tastes); // "délicieux"
console.log(good_morning.constructor); // Gaufre()

// antipattern: new a été oublié, mais l'objet fonctionne quand même
var good_morning = Gaufre();
console.log(typeof good_morning); // "object"
console.log(window.tastes); // "undefined"
console.log(good_morning.constructor); // Gaufre()
```

Au moins 2 façons de créer des classes sous ES5

On rappelle que, si l'on souhaite utiliser une fonction comme constructeur d'un objet, on écrira quelque chose du genre :

```
var monObjet = new MyColor();
```

Pour que cela fonctionne, il faut que les méthodes et propriétés publiques de cet objet soient préfixées par « this » dans le code source de la fonction MyColor().

Nous commençons par créer une fonction MyColor qui fait appel à une méthode interne « init » (on sait qu'il s'agit d'une méthode du fait de la présence du préfixe « this ») :

```
var MyColor = function () {
    this.init();
}
```

Nous définissons ensuite le code de la méthode « init » en l'associant à l'objet « prototype » de la fonction MyColor :

```
MyColor.prototype.init = function () {
    this.minSpeed = .6;
    this.maxSpeed = 1.8;
    // etc.. je ne détaille pas toute la méthode ici, sachant que vous
    // trouverez le code source complet plus loin dans ce chapitre
};
```

Quelle que soit son origine (fonction ou autre), tout objet Javascript a pour objet parent un objet « prototype », qu'il est possible de modifier en lui ajoutant des méthodes et propriétés spécifiques. Pour faire « court », disons que ces méthodes et propriétés « redescendront » par héritage sur tous les objets instanciés à partir de la fonction considérée (en l'occurrence la fonction MyColor).

Le mieux, pour bien comprendre le principe, c'est de le tester sur un cas concret, aussi voici le code source complet de l'objet MyColor :

```
var MyColor = function () {
    this.init();
}
MyColor.prototype.init = function () {
    this.minSpeed = .6;
    this.maxSpeed = 1.8;
    this.minR = 200;
    this.maxR = 255;
    this.minG = 20;
    this.maxG = 120;
    this.minB = 100;
    this.maxB = 140;
    this.R = random(this.minR, this.maxR);
    this.G = random(this.minG, this.maxG);
    this.B = random(this.minB, this.maxB);
    this.Rspeed = (random(1) > .5 ? 1 : -1) * random(this.minSpeed, this.maxSpeed);
    this.Gspeed = (random(1) > .5 ? 1 : -1) * random(this.minSpeed, this.maxSpeed);
    this.Bspeed = (random(1) > .5 ? 1 : -1) * random(this.minSpeed, this.maxSpeed);
};
MyColor.prototype.update = function () {
    this.Rspeed = (((this.R += this.Rspeed) > this.maxR) || (this.R < this.minR)) ?
        -this.Rspeed : this.Rspeed;
    this.Gspeed = (((this.G += this.Gspeed) > this.maxG) || (this.G < this.minG)) ?
        -this.Gspeed : this.Gspeed;
    this.Bspeed = (((this.B += this.Bspeed) > this.maxB) || (this.B < this.minB)) ?
        -this.Bspeed : this.Bspeed;
};
MyColor.prototype.getColor = function () {
    return color(this.R, this.G, this.B);
```

```
};
```

J'indiquais précédemment qu'il était possible de créer notre objet MyColor selon une autre méthode. La voici :

```
var MyColor_Proto = {
    minSpeed : .6,
    maxSpeed : 1.8,
    minR : 200,
    maxR : 255,
    minG : 20,
    maxG : 12,
    minB : 100,
    maxB : 140
};
var MyColor = function () {
    var that = Object.create(MyColor_Proto);
    that.R = random(that.minR, that.maxR);
    that.G = random(that.minG, that.maxG);
    that.B = random(that.minB, that.maxB);
    that.Rspeed = (random(1) > .5 ? 1 : -1) * random(that.minSpeed, that.maxSpeed);
    that.Gspeed = (random(1) > .5 ? 1 : -1) * random(that.minSpeed, that.maxSpeed);
    that.Bspeed = (random(1) > .5 ? 1 : -1) * random(that.minSpeed, that.maxSpeed);
    that.update = function () {
        that.Rspeed = (((that.R += that.Rspeed) > that.maxR) ||
                      (that.R < that.minR)) ? -that.Rspeed : that.Rspeed;
        that.Gspeed = (((that.G += that.Gspeed) > that.maxG) ||
                      (that.G < that.minG)) ? -that.Gspeed : that.Gspeed;
        that.Bspeed = (((that.B += that.Bspeed) > that.maxB) ||
                      (that.B < that.minB)) ? -that.Bspeed : that.Bspeed;
    };
    that.getColor = function () {
        return color(that.R, that.G, that.B);
    };
    return that ;
}
```

On commence par créer un objet que j'ai appelé « that » (par convention, et pour éviter tout risque de confusion avec « this »). Pour créer cet objet « that » j'ai utilisé la méthode « create » de l'objet « Object ». C'est un objet standard de Javascript, qui est en quelque sorte le « papa » de tous les autres objets.

La présence du « return that » à la fin de la fonction est essentielle, sans cela nous ne serons pas en mesure de récupérer l'objet généré par la fonction, et nous ne pourrons pas utiliser ses propriétés et méthodes. Je vous invite à mettre cette ligne en commentaire, et à regarder ce qui se passe.

On notera que la méthode `Object.create()` est apparue sous ES5. Il existe un polyfill (prothèse d'émulation) pour les navigateurs les plus anciens. On trouvera ce polyfill ici :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Object/create

3.10.2 Notion de modularité

On a vu au chapitre 3.9.7 une méthode permettant de définir une notion de visibilité publique/privée, mais nous avons vu également que cette méthode posait des problèmes de compatibilité « cross browser » et qu'elle rencontrait un succès mitigé.

Nous allons dans ce chapitre utiliser la technique des IIFE pour créer un objet modulaire contenant des méthodes privées et publiques. La méthode privée « `_init` » sera spécialisée dans l'initialisation des différentes méthodes publiques dont le nom commence par « `init` » et dont le module a besoin pour fonctionner.

Voici un squelette fonctionnel permettant d'identifier les grandes lignes de ce que je viens d'énoncer :

```
var skeletModule = (function () {
    function _init() {
        for (var func in this) {
            if (typeof this[func] == 'function'
                && func.substr(0, 4) == 'init' && func != 'init')
                this[func]();
        }
    }

    // Déclaration des méthodes et propriétés privées
    var CONST_TEST = 'hello world';
    function _PrivateFunctionXX (){
    }
    function _PrivateFunctionYY (){
    }
    function _PrivateFunctionZZ (){
    }
    function _PrivateFunctionWW (){
    }

    // Déclaration des méthodes et propriétés publiques
    return {
        author: 'xxx',
        version: '1.0.0',
        init: _init,
        initXX : _PrivateFunctionXX,
        initYY : _PrivateFunctionYY,
        calcData : _PrivateFunctionZZ,
    };
})();
```

On voit que la déclaration des méthodes et propriétés publiques se fait à l'intérieur du « return », les méthodes et propriétés qui précèdent ne sont accessibles à l'extérieur que dans le cas où on les a mises en correspondance avec un des éléments déclarés sous le « return ». Ainsi la méthode publique « calcData » est liée à la méthode privée « _PrivateFunctionZZ », cette dernière ne peut être appelée qu'au travers de la méthode « calcData » et pas directement. La méthode « _PrivateFunctionWW » n'est liée à aucune méthode publique, elle ne peut donc être utilisée que par les autres méthodes privées.

Dans l'exemple qui précède, les méthodes « _PrivateFunctionXX » et « _PrivateFunctionYY » seront appelées à l'initialisation du module, au travers des méthodes publiques « initXX » et « initYY ».

Cette manière d'initialiser le module n'est pas une obligation. Cette méthode ne permet pas, par exemple, de définir précisément l'ordre dans lequel les fonctions d'initialisation seront appelées. On peut éventuellement remplacer le code de la méthode « _init » par un code « en dur » appelant les méthodes à initialiser dans un ordre précis.

Voici un autre exemple, avec le module « protoModule », plus proche de la « vraie vie », avec des méthodes privées appelant d'autres méthodes privées (je vous laisse déterminer lesquelles) et utilisant des propriétés privées.

```
'use strict';

var protoModule = (function () {
    /**
     * Lancement de toutes les méthodes du module commençant par "init",
     * à l'exclusion de la méthode "init" elle-même
     * (qui doit être lancée manuellement)
     * @private
     */
    function _init() {
        for (var func in this) {
            if (typeof this[func] == 'function'
                && func.substr(0, 4) == 'init' && func != 'init')
                this[func]();
        }
    }

    // Déclaration des méthodes et propriétés privées

    var CONST_TEST = 'hello world'; // avec ES6 on peut utiliser "const"

    /**
     * Initialisation du sous-module LoadMapAPI
     * @param param1
     * @private
     */
    function _internalLoadMapAPI (param1) {
        if (param1 != undefined && param1 != '') {
            alert(param1);
        } else {
            alert('initLoadMapAPI');
        }
    }
})
```

```
/*
 * Initialisation du sous-module FormValidate
 * @private
 */
function _internalFormValidate () {
    alert('initFormValidate');
}

/**
 * Méthode de calcul de distance
 * @private
 */
function _calcDistance () {
    // exemple d'utilisation d'une propriété privée
    alert(CONST_TEST);
}

/**
 * Méthode de calcul de distance
 * @private
 */
function _internalDistanceCalc () {
    // exemple d'appel d'une méthode privée par une autre méthode privée
    _calcDistance();
}

// Déclaration des méthodes et propriétés publiques
return {
    author: 'xxx',
    version: '1.0.0',
    init: _init,
    initLoadMapAPI : _internalLoadMapAPI,
    initFormValidate : _internalFormValidate,
    calcDistance : _internalDistanceCalc
};
})();

window.addEventListener('load', function () {
    protoModule.init();

    protoModule.initLoadMapAPI('rappel de la méthode "initLoadMapAPI"');

    protoModule.calcDistance();
}, false) ;
```

3.10.3 Data Binding

Avec l'arrivée de frameworks JS comme Backbone, Knockout et Angular (vers 2009-2010) , on a vu arriver dans les pratiques de développement une nouvelle manière de concevoir les pages webs, s'appuyant sur un mécanisme appelé « data binding ». Ce mécanisme consiste à lier de manière automatique des données métiers à des éléments du DOM. Cette manière de développer a remis profondément en cause les pratiques utilisées jusqu'alors dans le développement des interfaces utilisateur.

On va voir dans ce chapitre qu'il est possible de recourir à un mécanisme de type « data binding » sans pour autant recourir à un framework de type Angular.

Ce chapitre tire beaucoup d'éléments d'un excellent article publié par Luca Ongaro en décembre 2012 :

<http://www.lucaongaro.eu/blog/2012/12/02/easy-two-way-data-binding-in-javascript/>

Dans son article, Luca Ongaro propose une implémentation pour JQuery et une version en VanillaJS. C'est cette dernière que nous allons étudier ici.

Dans l'exemple de code HTML suivant, nous avons 2 champs de saisie et 3 spans.

```
<div>Data Binding en Vanilla JS</div>
<input type="number" data-bind-123="name" /><br>
<span data-bind-123="name"></span><br>
<span data-bind-123="name"></span><br>
<span data-bind-123="name"></span><br>
<input type="number" data-bind-123="name" /><br>
```

Tous ces éléments sont liés entre eux au moyen d'un même attribut « data-bind-123 » qui a la valeur « name ». Si l'un de ces éléments avait une valeur différente, il ne serait plus lié aux autres.

Voici une copie d'écran du résultat produit par le code HTML dans un navigateur. Dans cet exemple, on s'est amusé alternativement l'un et l'autre champ de saisie, soit en cliquant sur les flèches, soit en modifiant manuellement l'une ou l'autre des valeurs. Dans tous les cas, les autres éléments ont vu leur contenu se synchroniser automatiquement :

Data Binding en Vanilla JS

21	▲
21	
21	
21	▼
21	

Comment cette synchronisation est-elle effectuée, c'est ce que nous allons voir dans la suite de ce chapitre.

Pour mettre en place ce mécanisme de « data binding » :

1. Nous avons besoin d'un moyen de préciser quels éléments d'interface utilisateur sont liés à quelles propriétés
2. Nous devons surveiller les changements qui interviennent sur les propriétés et sur les éléments de l'interface utilisateur
3. Nous devons propager tout changement détecté à tous les objets et éléments liés

Il existe de multiples façons d'y parvenir, mais une approche rapide et efficace consiste à utiliser le design pattern PubSub, tel qu'il présenté par Luca Ongaro dans son article. L'idée est simple, elle consiste à utiliser des attributs de données personnalisés pour spécifier les liaisons au niveau du code HTML. Tous les objets JavaScript et les éléments DOM qui sont liés ensemble vont "souscrire" à un objet PubSub. Chaque fois qu'un changement est détecté sur l'objet JavaScript ou sur un élément de saisie HTML, nous transmettons l'événement au PubSub, qui diffuse et propage le changement sur tous les autres objets et éléments liés.

La grande force de cette solution, c'est qu'un écouteur d'évènement de type « change » est placé sur un élément de niveau supérieur par rapport aux éléments surveillés (par exemple au niveau d'une « DIV » ou du nœud « document »). Cela évite d'avoir à placer un écouteur d'évènement sur chaque élément séparément.

On notera que PubSub est l'abréviation de Publish-Subscribe.

La définition proposée par Wikipédia pour ce design pattern est la suivante :

Publish-subscribe (littéralement : *publier-souscrire*) est un mécanisme de publication et de souscription de messages dans lequel les émetteurs (*publisher*, littéralement : *éditeurs*) ne destinent pas a priori les messages à des destinataires (*subscriber*, littéralement : *abonné*). À la place, une catégorie est associée aux messages émis sans savoir s'il y a des destinataires. De la même manière, les destinataires souscrivent aux catégories les intéressent, et ne reçoivent que les messages correspondant, sans savoir s'il y a des émetteurs.

Voici quelques articles intéressants relatifs à ce sujet :

https://en.wikipedia.org/wiki/Publish%20-%20subscribe_pattern

<http://www.svlada.com/publish-subcribe-pattern-javascript/>

Pour clarifier notre propos, voici le code source de notre page HTML de test :

```
<!DOCTYPE html>
<html lang="fr">
    <head>
        <title>Data Binding in Vanilla JS</title>
        <meta charset="UTF-8">
        <script src="databinder.js"></script>
        <script src="model/user.js"></script>
    </head>
    <body>
        <div>Data Binding en Vanilla JS</div>
        <input type="number" data-bind-123="name" /><br>
        <span data-bind-123="name"></span><br>
        <span data-bind-123="name"></span><br>
        <span data-bind-123="name"></span><br>
        <input type="number" data-bind-123="name" /><br>
        <script>
            document.body.onload = function () {
                var user = new User(123);
                user.set("name", "Wolfgang");
            }
        </script>
    </body>
</html>
```

Le data binding est piloté par un objet Javascript, dont voici le code source :

```
/*
 * Easy Two-Way Data Binding in JavaScript
 * by Luca Ongaro Dec 2nd, 2012
 * http://www.lucaongaro.eu/blog/2012/12/02/easy-two-way-data-binding-in-javascript/
 * Added "target_id" parameter to target other DOM elements than "document"
 */
function DataBinder(object_id, target_id) {
    // Create a simple PubSub object
    var pubSub = {
        callbacks: {},
        on: function (msg, callback) {
            this.callbacks[msg] = this.callbacks[msg] || [];
            this.callbacks[msg].push(callback);
        },
        publish: function (msg) {
            this.callbacks[msg] = this.callbacks[msg] || []
            for (var i = 0, len = this.callbacks[msg].length; i < len; i++) {
                this.callbacks[msg][i].apply(this, arguments);
            }
        }
    };
    var data_attr = "data-bind-" + object_id;
    var message = object_id + ":change";
    var changeHandler = function (evt) {
        var target = evt.target || evt.srcElement; // IE8 compatibility
        var prop_name = target.getAttribute(data_attr);
        if (prop_name && prop_name !== "") {
            pubSub.publish(message, prop_name, target.value);
        }
    }
}
```

```

        }
    };

    // Listen to change events and proxy to PubSub
    if (target_id != undefined && target_id != '') {
        var target_item = document.getElementById(target_id) ;
        if (target_item) {
            if (target_item.addEventListener) {
                target_item.addEventListener("change", changeHandler, false);
            } else {
                // IE8 uses attachEvent instead of addEventListener
                target_item.attachEvent("onchange", changeHandler);
            }
        }
    } else {
        if (document.addEventListener) {
            document.addEventListener("change", changeHandler, false);
        } else {
            // IE8 uses attachEvent instead of addEventListener
            document.attachEvent("onchange", changeHandler);
        }
    }
}

// PubSub propagates changes to all bound elements
pubSub.on(message, function (evt, prop_name, new_val) {
    var elements = document.querySelectorAll("[ " + data_attr + "=" +
        prop_name + " ]") ;
    var tag_name;

    for (var i = 0, len = elements.length; i < len; i++) {
        tag_name = elements[i].tagName.toLowerCase();

        if (tag_name === "input" || tag_name === "textarea"
            || tag_name === "select") {
            elements[i].value = new_val;
        } else {
            elements[i].innerHTML = new_val;
        }
    }
});

return pubSub;
}

```

Par rapport à la version initiale proposée par Luca Ongaro, j'ai ajouté la possibilité d'affecter l'écouteur d'évènement à une autre cible du DOM que l'élément « document ». Cela se fait via le nouveau paramètre « target_id ».

Nous avons aussi besoin d'un modèle, ici ce sera le modèle User :

```
function User(uid, target_id) {
    var binder = new DataBinder(uid, target_id),
        user = {
            attributes: {},
            // The attribute setter publish changes using the DataBinder PubSub
            set: function (attr_name, val) {
                this.attributes[attr_name] = val;
                // Use the `publish` method
                binder.publish(uid + ":change", attr_name, val, this);
            },
            get: function (attr_name) {
                return this.attributes[attr_name];
            },
            getAll: function () {
                return this.attributes;
            },
            _binder: binder
        };
    // Subscribe to the PubSub
    binder.on(uid + ":change", function (evt, attr_name, new_val, initiator) {
        if (initiator !== user) {
            user.set(attr_name, new_val);
        }
    });
    return user;
}
```

Par rapport à la version proposée par Luca Ongaro, j'ai ajouté le paramètre « target_id » permettant de définir une cible autre que l'élément « document ». J'ai également ajouté une méthode « getAll » permettant de renvoyer la totalité des attributs en une seule fois.

3.11 *Templating*

3.11.1 ES6

ES6 apporte au développeur la possibilité de définir des littéraux de gabarit :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Litt%C3%A9raux_gabarits

Cette possibilité nouvelle n'est pas supportée par les navigateurs les plus anciens, d'où un succès mitigé, ou en tout cas un démarrage plutôt lent.

Voici un exemple emprunté à la documentation de Mozilla montrant qu'on a la possibilité d'interpoler des variables à l'intérieur de chaînes de caractères :

```
var a = 5;
var b = 10;
console.log(`Quinze est ${a + b} et n'est pas ${2 * a + b}.`);
// "Quinze est 15 et n'est pas 20."
```

TODO : ce chapitre devra être approfondi ultérieurement

3.11.2 Handlebars.js

HandlebarsJS est un puissant moteur de templating JS. Conçu en premier lieu pour le framework EmberJS, HandlebarsJS peut être utilisé indépendamment de EmberJS, comme nous allons le voir dans un exemple.

Le site officiel :

<http://handlebarsjs.com/>

On peut télécharger localement Handlebars, mais on peut aussi le charger à partir de cdnjs :

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.6/handlebars.min.js"></script>
```

Voici un exemple de tableau HTML dont le contenu est généré par Handlebars à partir d'un template et d'un jeu de données au format JSON :

Id Voyage	Libellé	Id Chauffeur	Nom Chauffeur	Nb Liv.	Date départ
1	Voyage 1	CHF001	ANDRE Martin	10	2016/01/23
2	Voyage 2	CHF003	DANIEL André	3	2016/01/23
3	Voyage 3	CHF020	MICHAUD Henri	5	2016/01/23
4	Voyage 4	CHF002	RIAD Hassan	1	2016/01/23
5	Voyage 5	CHF001	ANDRE Martin	0	2016/01/23
6	Voyage 6	CHF008	SCHMIDT Heinrik	4	2016/01/23

Le tableau HTML inséré dans la page est un tableau vide. On notera la présence d'un attribut ID sur la balise « tbody » :

```
<table border="1px" cellspacing="0" width="100%">
  <thead>
    <tr>
      <th>Id Voyage</th>
      <th>Libellé</th>
      <th>Id Chauffeur</th>
      <th>Nom Chauffeur</th>
      <th>Nb Liv.</th>
      <th>Date départ</th>
    </tr>
  </thead>
  <tbody id="datalist_voy">
    <tr data-id="">
      <td>&nbsp;</td>
      <td>&nbsp;</td>
      <td>&nbsp;</td>
      <td>&nbsp;</td>
      <td>&nbsp;</td>
      <td>&nbsp;</td>
    </tr>
  </tbody>
</table>
```

Le template est un script avec un type particulier. Celui-ci va nous servir à regénérer la partie « tbody » du tableau HTML :

```
<script id="voyage_tmpl" type="text/x-handlebars-template">
  <tbody>
    {{#each messages}}
    <tr class="sort" data-id="{{id}}>
      <td>{{id}}</td>
      <td>{{libelle}}</td>
      <td>{{idchf}}</td>
      <td>{{nomchf}}</td>
      <td>{{nbliv}}</td>
      <td>{{startdate}}</td>
    </tr>
    {{/each}}
  </tbody>
</script>
```

Le jeu de données JSON est ici défini en dur, mais il aurait pu être chargé à partir d'une requête AJAX :

```
var voyages = {messages:[
    {"id":"1","libelle":"Voyage 1","idchf":"CHF001","nomchf":"ANDRE
Martin","nbliv": "10", "startdate": "2016/01/23", "divers": "" },
    {"id": "2", "libelle": "Voyage 2", "idchf": "CHF003", "nomchf": "DANIEL
André", "nbliv": "3", "startdate": "2016/01/23", "divers": "" },
    {"id": "3", "libelle": "Voyage 3", "idchf": "CHF020", "nomchf": "MICHAUD
Henri", "nbliv": "5", "startdate": "2016/01/23", "divers": "" },
    {"id": "4", "libelle": "Voyage 4", "idchf": "CHF002", "nomchf": "RIAD
Hassan", "nbliv": "1", "startdate": "2016/01/23", "divers": "" },
    {"id": "5", "libelle": "Voyage 5", "idchf": "CHF001", "nomchf": "ANDRE
Martin", "nbliv": "0", "startdate": "2016/01/23", "divers": "" },
    {"id": "6", "libelle": "Voyage 6", "idchf": "CHF008", "nomchf": "SCHMIDT
Heinrik", "nbliv": "4", "startdate": "2016/01/23", "divers": "" }
]};
```

Le principe consiste à récupérer le contenu du template, à le passer à la méthode « compile » de l'objet « Handlebars ». Cette compilation génère un objet, auquel on transmettra le jeu de données JSON. L'objet nous renverra en sortie un code HTML que nous pourrons placer à un endroit de la page (dans notre exemple on va remplacer la partie « tbody » du tableau HTML) :

```
window.addEventListener('load', function() {
    var tmplsrc = document.getElementById('voyage_tmpl').innerHTML;
    var template = Handlebars.compile(tmplsrc);

    var datalist_voy = document.getElementById("datalist_voy");
    if (datalist_voy) {
        var result = template(voyages);
        datalist_voy.innerHTML = result;
    } else {
        console.log('Error : item "datalist_voy" not found');
    }

}, false);
```

3.11.3 Hyperscript

Hyperscript constitue une manière originale de générer du HTML.

C'est un peu le challenger que personne n'attendait, mais son utilisation est pour l'instant limitée à quelques frameworks confidentiels comme MithrilJS. Le principe de Hyperscript est de générer un DOM virtuel à partir de pur code JS, et de générer le HTML à partir de ce DOM virtuel. Les performances sont excellentes.

Voici un exemple de page HTML piloté par le framework Mithril et dont le code HTML est produit par Hyperscript :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello world</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <div id="link1"></div>
    <script src="../mithril/mithril.js"></script>
    <script src="index09.js"></script>
  </body>
</html>
```

Le code source généré par Hyperscript :

```
<div>
  <label>Name:</label><input placeholder="What is your name?" type="text">
  <p class="output">greg</p>
</div>
```

Du côté de l'internaute, ça donne ça :

Name:

greg

Le code source du fichier index09.js :

```
"use strict";

var myData = {};
myData.yourName = 'greg'; // Piece of data

// Plain event handler
function handleNameInput(evt) {
    myData.yourName = evt.currentTarget.value;
    console.log(myData.yourName);
    var temp = document.querySelector('.output');
    console.dir(temp);
    if (temp != undefined) {
        temp.innerHTML = myData.yourName;
    }
}

// This function uses the 'hyperscript' notation to create the virtual DOM.
function renderMaquette() {
    return m('div', [
        m('label', "Name:"),
        m('input', {
            type: 'text',
            placeholder: 'What is your name?',
            value: myData.yourName,
            onchange: handleNameInput
        }),
        m('p.output', ['Hello ' + (myData.yourName || 'you') + '!'])
    ]);
}

m.render(document.querySelector('#link1'), renderMaquette());
```

Quelques sites de référence :

<https://mithril.js.org/>

<https://github.com/hyperhype/hyperscript>

Quelques sites complémentaires, dont plusieurs outils en ligne permettant de générer du code Hyperscript à partir de code HTML :

- [html2hscrip](#) - Parse HTML into hyperscript
- [dom2hscrip](#) - Frontend library for parsing HTML into hyperscript using the browser's built in parser.
- [html2hscrip.herokuapp.com](#) - Online Tool that converts html snippets to hyperscript
- [html2hyperscript](#) - Original commandline utility to convert legacy HTML markup into hyperscript
- [hyperscript-helpers](#) - write `div(h1('hello'))` instead of `h('div', h('h1', 'hello'))`
- [react-hyperscript](#) - use hyperscript with React.

4 Document Object Model (DOM)

4.1 Comprendre le DOM

4.1.1 Introduction

Un document HTML est constitué d'éléments HTML imbriqués les uns dans les autres, que l'on représente généralement sous la forme d'un arbre.

Le Document Object Model (DOM) est un arbre hiérarchique composé de noeuds qui sont autant d'objets d'un point de vue du langage Javascript.

Le DOM avait à l'origine été conçu pour servir d'interface de programmation pour la manipulation de documents XML. HTML et XML ayant des racines communes, il apparaissait assez naturel aux développeurs des premiers navigateurs de s'appuyer sur cette interface de programmation pour représenter la structure d'une page HTML dans le moteur des navigateurs.

Quand le navigateur charge une page HTML, il la "parse" et l'analyse pour s'en faire une représentation interne basée sur le DOM.

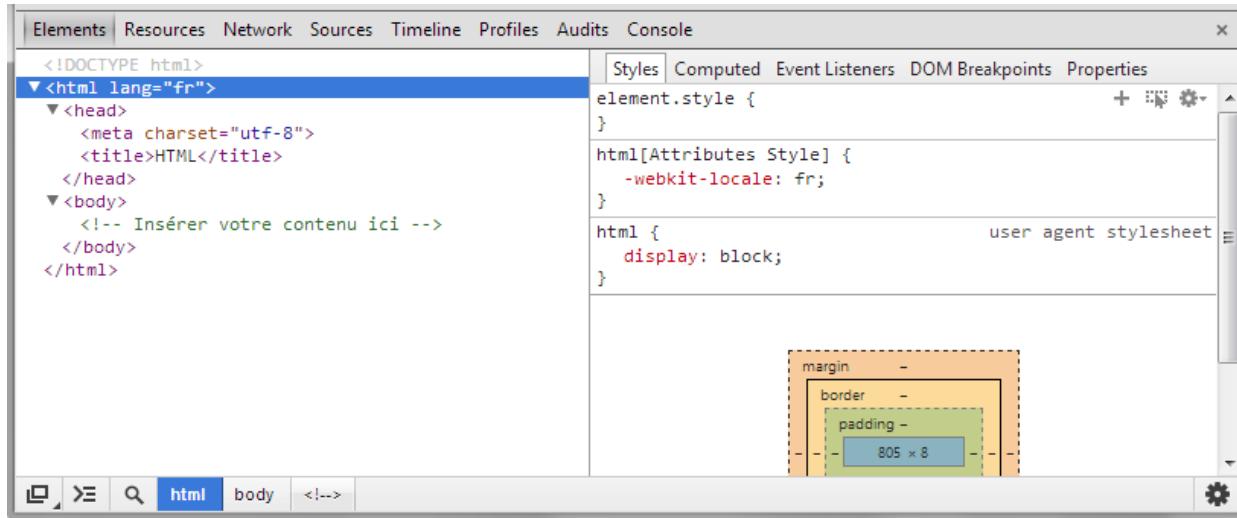
Prenons pour exemple la page HTML suivante :

```
<!DOCTYPE html>
<html lang="fr">
<head>
<meta charset="utf-8">
<title>HTML</title>
</head>
<body>
<!-- Insérer votre contenu ici -->
</body>
</html>
```

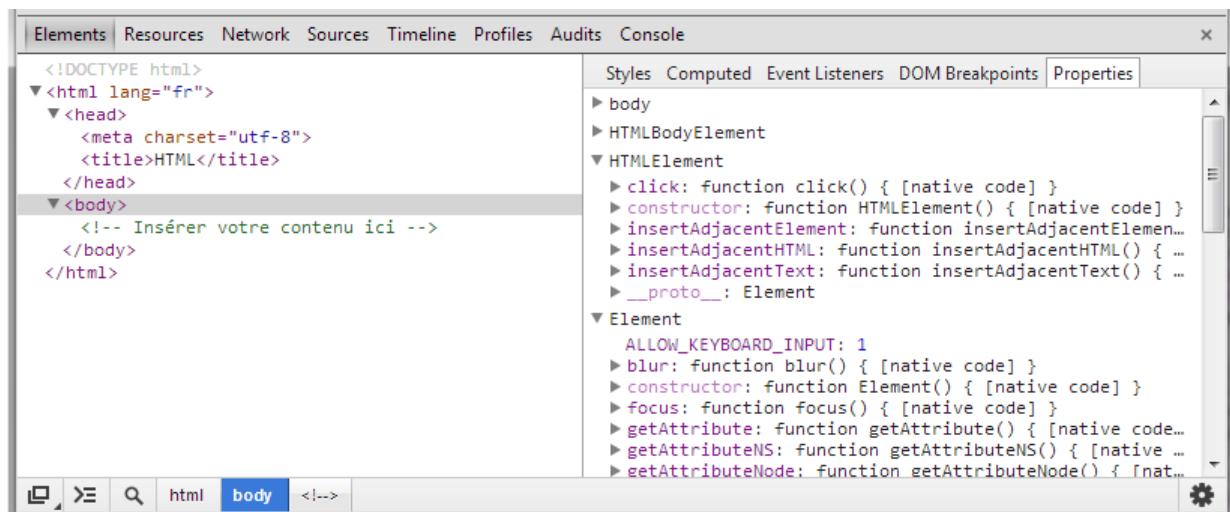
IMPORTANT : il est important de savoir que le code source HTML affiché par le navigateur est un code source figé qui correspond au code HTML chargé initialement pour une page donnée. Les outils de débogage offrent des facilités pour consulter le DOM sous sa forme HTML, mais ce code peut différer du code HTML initial, car le DOM peut être largement modifié - et à tout moment - par du code Javascript.

Exemples de représentation du DOM via les outils de développement de Google Chrome (activable via la touche F12) :

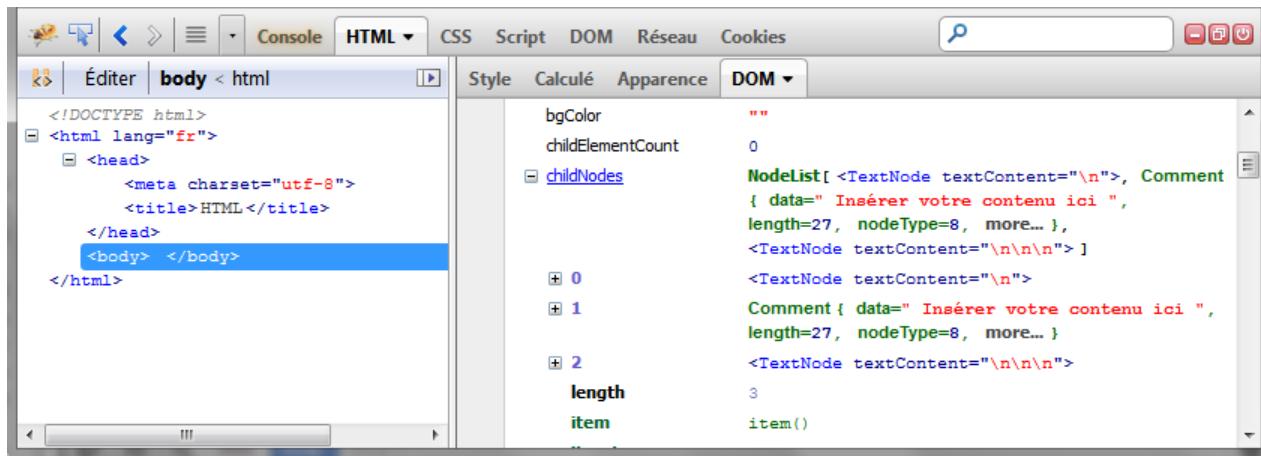
- exemple 1 (affichage des styles associés à l'élément sélectionné, dans la partie droite de l'écran) :



- exemple 2 (sélection de l'onglet "propriétés" dans la partie de droite) :



La même page affichée dans le plugin Firebug (associé à Firefox) :



L'élément `<body>` sélectionné est un élément neud, et une instance de l'interface `HTMLBodyElement` (sur laquelle nous reviendrons ultérieurement).

L'objectif du DOM est de fournir une interface de programmation permettant d'interagir précisément avec les différents éléments composant la page HTML.

4.1.2 Les noeuds

Voici une liste des types de neuds les plus courants rencontrés dans les documents HTML :

- DOCUMENT_NODE (par exemple : `window.document`)
- ELEMENT_NODE (par exemple : `<body>`, `<a>`, `<p>`, `<script>`, `<style>`, `<html>`, `<h1>`)
- ATTRIBUTE_NODE (par exemple : l'attribut "class", ou encore l'attribut "required")
- TEXT_NODE (par exemple, les caractères de texte dans un document HTML, y compris les retours chariot et les espaces)
- DOCUMENT_FRAGMENT_NODE (par exemple : `document.createDocumentFragment()`)
- DOCUMENT_TYPE_NODE (par exemple : `<!DOCTYPE html>`)

Les types de neuds ci-dessus sont codifiés exactement tels qu'ils sont définis dans les propriétés de l'objet `Node`, qui fait partie intégrante de l'environnement de votre navigateur. Les propriétés définies dans l'objet `Node` sont des constantes utilisées en interne par le navigateur. Le développeur n'a en règle générale pas à se soucier de la valeur de ces constantes, mais il est bon de savoir ce qui se cache sous le capot.

Le code ci-dessous permet de lister l'ensemble des types de neuds définis dans le navigateur, et la valeur associée :

```

<!DOCTYPE html>
<html lang="fr">
<body>
    <script>
        for ( var key in Node) {
            console.log(key, ' = ' + Node[key]);
        };

        /*
        ELEMENT_NODE = 1
        ATTRIBUTE_NODE = 2
        TEXT_NODE = 3
        CDATA_SECTION_NODE = 4
        ENTITY_REFERENCE_NODE = 5
        ENTITY_NODE = 6
        PROCESSING_INSTRUCTION_NODE = 7
        COMMENT_NODE = 8
        DOCUMENT_NODE = 9
        DOCUMENT_TYPE_NODE = 10
        DOCUMENT_FRAGMENT_NODE = 11
        NOTATION_NODE = 12
        DOCUMENT_POSITION_DISCONNECTED = 1
        DOCUMENT_POSITION_PRECEDING = 2
        DOCUMENT_POSITION_FOLLOWING = 4
        DOCUMENT_POSITION_CONTAINS = 8
        DOCUMENT_POSITION_CONTAINED_BY = 16
        DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC = 32 */
    </script>
</body>
</html>

```

On peut exécuter ce code via la console du navigateur si on le souhaite.

4.1.3 L'objet Navigator

L'objet navigator est un objet standard défini dans l'environnement de tout navigateur.
On peut afficher ses propriétés au moyen du code suivant :

```

<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>

<div id="example"></div>

<script>

txt = "<p>Browser CodeName: " + navigator.appCodeName + "</p>";


```

```

txt+= "<p>Browser Name: " + navigator.appName + "</p>";
txt+= "<p>Browser Version: " + navigator.appVersion + "</p>";
txt+= "<p>Cookies Enabled: " + navigator.cookieEnabled + "</p>";
txt+= "<p>Browser Language: " + navigator.language + "</p>";
txt+= "<p>Browser Online: " + navigator.onLine + "</p>";
txt+= "<p>Platform: " + navigator.platform + "</p>";
txt+= "<p>User-agent header: " + navigator.userAgent + "</p>";
txt+= "<p>User-agent language: " + navigator.systemLanguage + "</p>";

document.getElementById("example").innerHTML=txt;

</script>

</body>
</html>

```

Attention : les informations renvoyées par l'objet navigator sont sujettes à caution. Elles sont peu fiables et il vaut mieux ne pas chercher à les utiliser pour détecter les caractéristiques de tel ou tel navigateur.

Par exemple Firefox renvoie les informations suivantes :

```

Browser CodeName: Mozilla
Browser Name: Netscape
Browser Version: 5.0 (Windows)
Cookies Enabled: true
Browser Language: fr
Browser Online: true
Platform: Win32
User-agent header: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:26.0) Gecko/20100101 Firefox/26.0
User-agent language: undefined

```

Google Chrome renvoie les informations suivantes :

```

Browser CodeName: Mozilla
Browser Name: Netscape
Browser Version: 5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/32.0.1700.76 Safari/537.36
Cookies Enabled: true
Browser Language: fr
Browser Online: true
Platform: Win32
User-agent header: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
    Chrome/32.0.1700.76 Safari/537.36
User-agent language: undefined

```

Et IE9 renvoie les informations suivantes :

```

Browser CodeName: Mozilla
Browser Name: Microsoft Internet Explorer
Browser Version: 5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; SLCC2; .NET CLR
    2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; InfoPath.2)
Cookies Enabled: true
Browser Language: undefined

```

Browser Online: true
 Platform: Win32
 User-agent header: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; .NET4.0C; InfoPath.2)
 User-agent language: fr

Point important :

On voit donc que l'objet Navigator est un "faux ami". Il ne constitue pas une solution fiable pour déterminer sur quel navigateur on se trouve.

En règle générale, il vaut mieux ne pas chercher à savoir sur quel navigateur on se trouve spécifiquement. Il est préférable de chercher à savoir si une fonctionnalité est disponible sur le navigateur courant (on verra des exemples tout au long du cours, et en particulier lors de l'utilisation des API HTML5).

Le site W3Schools propose un tableau de synthèse des propriétés et méthodes associées à l'objet Navigator :

Property	Description
appCodeName	Returns the code name of the browser
appName	Returns the name of the browser
appVersion	Returns the version information of the browser
cookieEnabled	Determines whether cookies are enabled in the browser
geolocation	Returns a Geolocation object that can be used to locate the user's position
language	Returns the language of the browser
onLine	Determines whether the browser is online
platform	Returns for which platform the browser is compiled
product	Returns the engine name of the browser
userAgent	Returns the user-agent header sent by the browser to the server
Method	Description
javaEnabled()	Specifies whether or not the browser has Java enabled
taintEnabled()	Removed in JavaScript version 1.2. Specifies whether the browser has data tainting enabled

On notera que l'objet Window fournit une propriété "navigator" qui est en fait un raccourci vers l'objet Navigator.

4.1.4 Objets "sub-nodes" hérités de Node

Chaque nœud de type objet dans un arbre DOM hérite des propriétés et méthodes de l'objet Node.

Selon le type de noeud du document, il existe également des "sous-noeuds" supplémentaires, que l'on désigne par le terme de "objets interfaces", sous-noeuds qui étendent l'objet Node. La liste suivante détaille le modèle d'héritage qui est mis en place par les navigateurs pour les interfaces de noeuds les plus courants (le symbole "<" indique "Hérité de") :

- Object < Node < Element < HTMLElement < (par exemple : HTML*Element)
- Object < Node < Attr (*déprécié dans DOM4*)
- Object < Node < CharacterData < Text
- Object < Node < Document < HTMLDocument
- Object < Node < DocumentFragment

L'objet Node est en fait une fonction "constructeur" de Javascript, qui hérite de l'objet "prototype", comme tous les objets créés en Javascript.

Pour vérifier que tous les noeuds héritent des propriétés et méthodes de l'objet Node, jetons un coup d'oeil à un élément HTML pris au hasard (en l'occurrence la balise `<a>`) et aux différentes propriétés et méthodes qui sont rattachées à l'objet sous-jacent de cet élément HTML :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <a href="#">Hi</a>

    <script>
        // on prend une référence sur la seule balise <a> de la page
        var nodeAnchor = document.querySelector('a');
        /* on crée un tableau "props" dans lequel on va stocker toutes les
           propriétés de l'objet sous-jacent à nodeAnchor */
        var props = [];
        /*
            on boucle sur les éléments de nodeAnchor pour collecter les
            propriétés et méthodes qui lui sont rattachées (y compris les
            propriétés et méthodes "héritées")
        */
        for ( var key in nodeAnchor) {
            props.push(key);
        }
        /* affichage de la console du tableau props (avec propriétés et méthodes
           triées alphabétiquement) */
        console.log(props.sort());
    </script>
</body>

</html>
```

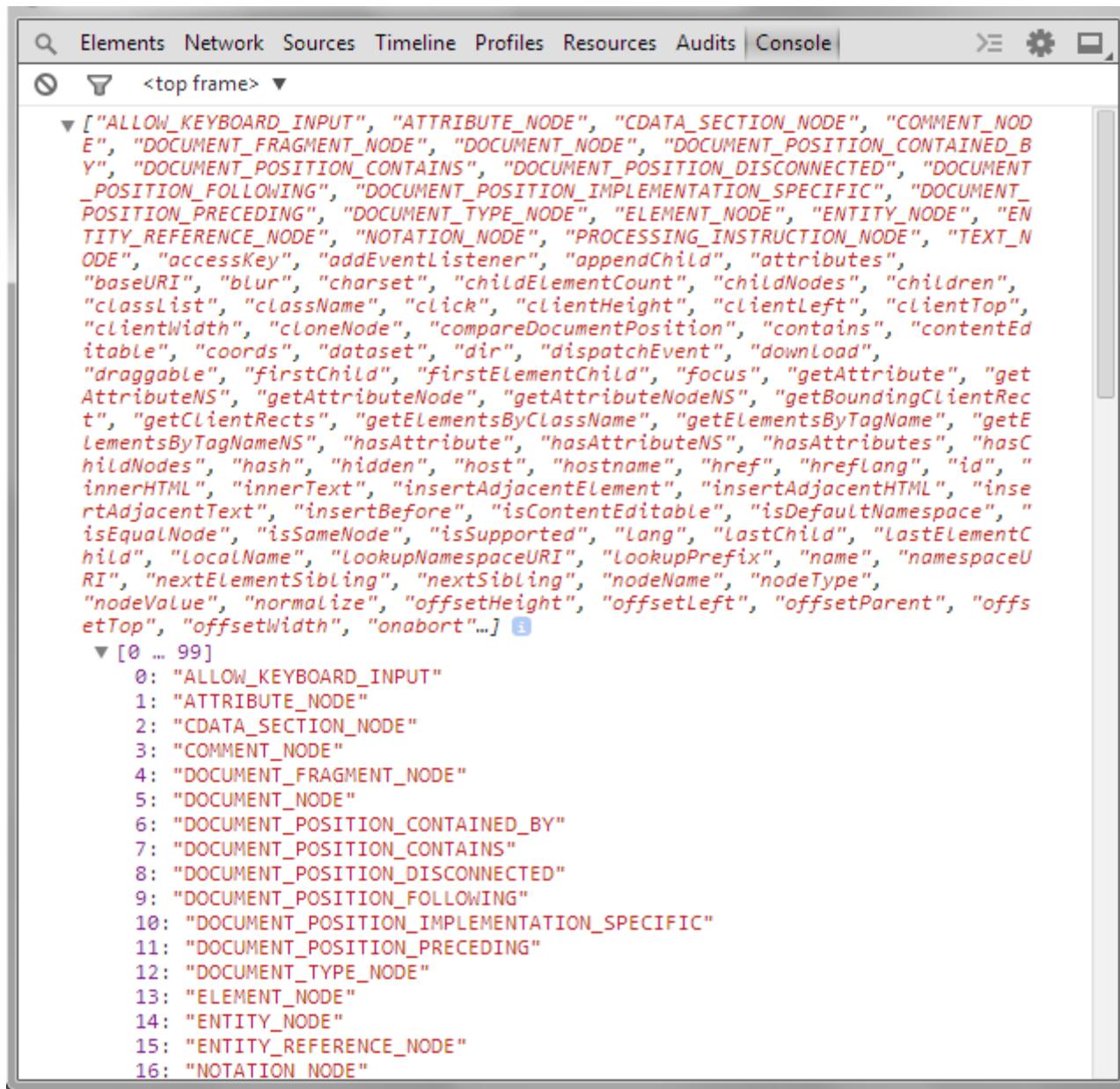
Passons un peu de temps dans l'outil de développement de Google Chrome pour regarder ce qui se passe dans les coulisses du navigateur.

Observons tout d'abord les propriétés de la balise `<a>` :

The screenshot shows the Google Chrome DevTools Elements tab. On the left, the DOM tree displays a script block containing code that creates an anchor element with a href of "#". The anchor element is selected. On the right, the Properties panel shows the prototype chain starting from `a`, then `HTMLAnchorElement`, then `HTMLElement`, and finally `Object`. Other properties like `constructor`, `toString`, and `__proto__` are also listed. At the bottom, a navigation bar shows `html`, `body`, and `a`, with `a` being the active tab.

On notera que l'objet sous-jacent de la balise `<a>` a hérité des propriétés et méthodes des objets suivants : `Element`, `HTMLElement`, `HTMLAnchorElement`, `Node`, et `Object`.

On peut également consulter ces différentes propriétés et méthodes au travers du tableau "props" pour lequel nous avons envoyé une photographie du contenu dans la log :



The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The output area displays a large array of DOM node types, each represented by a string. The array is divided into two sections: a long list at the top and a numbered list from 0 to 99 at the bottom. The strings in the array are color-coded in red, indicating they are likely URLs or specific node types.

```
["ALLOW_KEYBOARD_INPUT", "ATTRIBUTE_NODE", "CDATA_SECTION_NODE", "COMMENT_NODE", "DOCUMENT_FRAGMENT_NODE", "DOCUMENT_NODE", "DOCUMENT_POSITION_CONTAINED_BY", "DOCUMENT_POSITION_CONTAINS", "DOCUMENT_POSITION_DISCONNECTED", "DOCUMENT_POSITION_FOLLOWING", "DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC", "DOCUMENT_POSITION_PRECEDING", "DOCUMENT_TYPE_NODE", "ELEMENT_NODE", "ENTITY_NODE", "ENTITY_REFERENCE_NODE", "NOTATION_NODE", "PROCESSING_INSTRUCTION_NODE", "TEXT_NODE", "accessKey", "addEventlistener", "appendChild", "attributes", "baseURI", "blur", "charset", "childElementCount", "childNodes", "children", "classList", "className", "click", "clientHeight", "clientLeft", "clientTop", "clientWidth", "cloneNode", "compareDocumentPosition", "contains", "contentEditable", "coords", "dataset", "dir", "dispatchEvent", "download", "draggable", "firstChild", "firstElementChild", "focus", "getAttribute", "getAttributeNS", "getAttributeNode", "getAttributeNodeNS", "getBoundingClientRect", "getClientRects", "getElementsByClassName", "getElementsByTagName", "getElementsByTagNameNS", "hasAttribute", "hasAttributeNS", "hasAttributes", "hasChildNodes", "hash", "hidden", "host", "hostname", "href", "hreflang", "id", "innerHTML", "innerText", "insertAdjacentElement", "insertAdjacentHTML", "insertAdjacentText", "insertBefore", "isContentEditable", "isDefaultNamespace", "isEqualNode", "isSameNode", "isSupported", "lang", "lastChild", "lastElementChild", "localName", "lookupNamespaceURI", "lookupPrefix", "name", "namespaceURI", "nextElementSibling", "nextSibling", "nodeName", "nodeType", "nodeValue", "normalize", "offsetHeight", "offsetLeft", "offsetParent", "offsetTop", "offsetWidth", "onabort"]  
[0 ... 99]  
0: "ALLOW_KEYBOARD_INPUT"  
1: "ATTRIBUTE_NODE"  
2: "CDATA_SECTION_NODE"  
3: "COMMENT_NODE"  
4: "DOCUMENT_FRAGMENT_NODE"  
5: "DOCUMENT_NODE"  
6: "DOCUMENT_POSITION_CONTAINED_BY"  
7: "DOCUMENT_POSITION_CONTAINS"  
8: "DOCUMENT_POSITION_DISCONNECTED"  
9: "DOCUMENT_POSITION_FOLLOWING"  
10: "DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC"  
11: "DOCUMENT_POSITION_PRECEDING"  
12: "DOCUMENT_TYPE_NODE"  
13: "ELEMENT_NODE"  
14: "ENTITY_NODE"  
15: "ENTITY_REFERENCE_NODE"  
16: "NOTATION_NODE"
```

4.1.5 Propriétés et méthodes pour travailler avec le DOM

Après avoir étudié brièvement l'objet Node et les "sous-noeuds", voyons de quelles propriétés et méthodes nous disposons pour travailler avec le DOM, et en particulier avec ses noeuds et sous-noeuds.

Propriétés "noeuds" :

- childNodes

- firstChild
- lastChild
- nextSibling
- nodeName
- nodeType
- nodeValue
- parentNode
- previousSibling

Méthodes "noeuds" :

- appendChild()
- cloneNode()
- compareDocumentPosition()
- contains()
- hasChildNodes()
- insertBefore()
- isEqualNode()
- removeChild()
- replaceChild()

Méthodes "document" :

- document.createElement()
- document.createTextNode()

*Propriétés liées aux HTML*Element :*

- innerHTML
- outerHTML
- textContent
- innerText
- outerText
- firstElementChild
- lastElementChild
- nextElementChild
- previousElementChild
- children

*Méthodes liées aux HTML*Element :*

- insertAdjacentHTML()
- insertAdjacentText() (tous navigateurs sauf Firefox)

4.1.6 Identifier le type et le nom d'un noeud

Jouons un peu avec les propriétés et méthodes en affichant le contenu et/ou le nom de certaines d'entre elles dans la log du navigateur :

```
<!DOCTYPE html>
```

```
<html lang="fr">
<meta charset="utf-8">
<body>
    <a href="#">Hi</a>
    <script>
        /* C'est un DOCUMENT_TYPE_NODE (nodeType 10) parce que
           Node.DOCUMENT_TYPE_NODE === 10 */
        console.log('document.doctype => ', document.doctype);
        console.log('document.doctype.nodeName => ', document.doctype.nodeName);
        console.log('document.doctype.nodeType => ', document.doctype.nodeType);
        //logs 10 which maps to DOCUMENT_TYPE_NODE

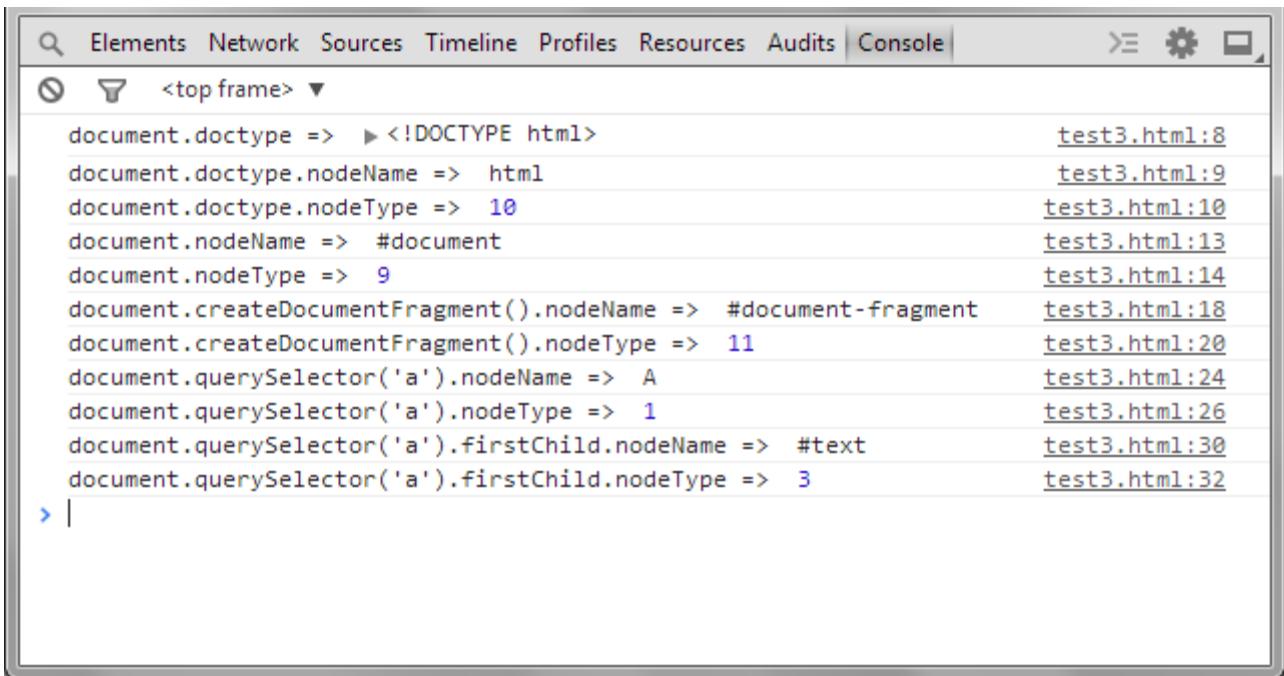
        // C'est un DOCUMENT_NODE (nodeType 9) parce que Node.DOCUMENT_NODE==9
        console.log('document.nodeName => ', document.nodeName);
        //logs '#document'
        console.log('document.nodeType => ', document.nodeType);
        //logs 9 which maps to DOCUMENT_NODE

        /* C'est un DOCUMENT_FRAGMENT_NODE (nodeType 11) parce que
           Node.DOCUMENT_FRAGMENT_NODE === 11 */
        console.log('document.createDocumentFragment().nodeName => ',
                   document.createDocumentFragment().nodeName);
        // logs '#document-fragment'
        console.log('document.createDocumentFragment().NodeType => ',
                   document.createDocumentFragment().nodeType);
        /* logs 11 ce qui correspond à DOCUMENT_FRAGMENT_NODE */

        //This is ELEMENT_NODE or nodeType 1 because Node. ELEMENT_NODE === 1
        console.log("document.querySelector('a').nodeName => ",
                   document.querySelector('a').nodeName); //logs 'A'
        console.log("document.querySelector('a').nodeType => ",
                   document.querySelector('a').nodeType);
        //logs 1 which maps to ELEMENT_NODE

        //This is TEXT_NODE or nodeType 3 because Node.TEXT_NODE === 3
        console.log("document.querySelector('a').firstChild.nodeName => ",
                   document.querySelector('a').firstChild.nodeName);
        //logs '#text'
        console.log("document.querySelector('a').firstChild.nodeType => ",
                   document.querySelector('a').firstChild.nodeType);
        /* logs 3 which maps to TEXT_NODE */
    </script>
</body>
</html>
```

Dans la console de Google Chrome, on obtient ceci :



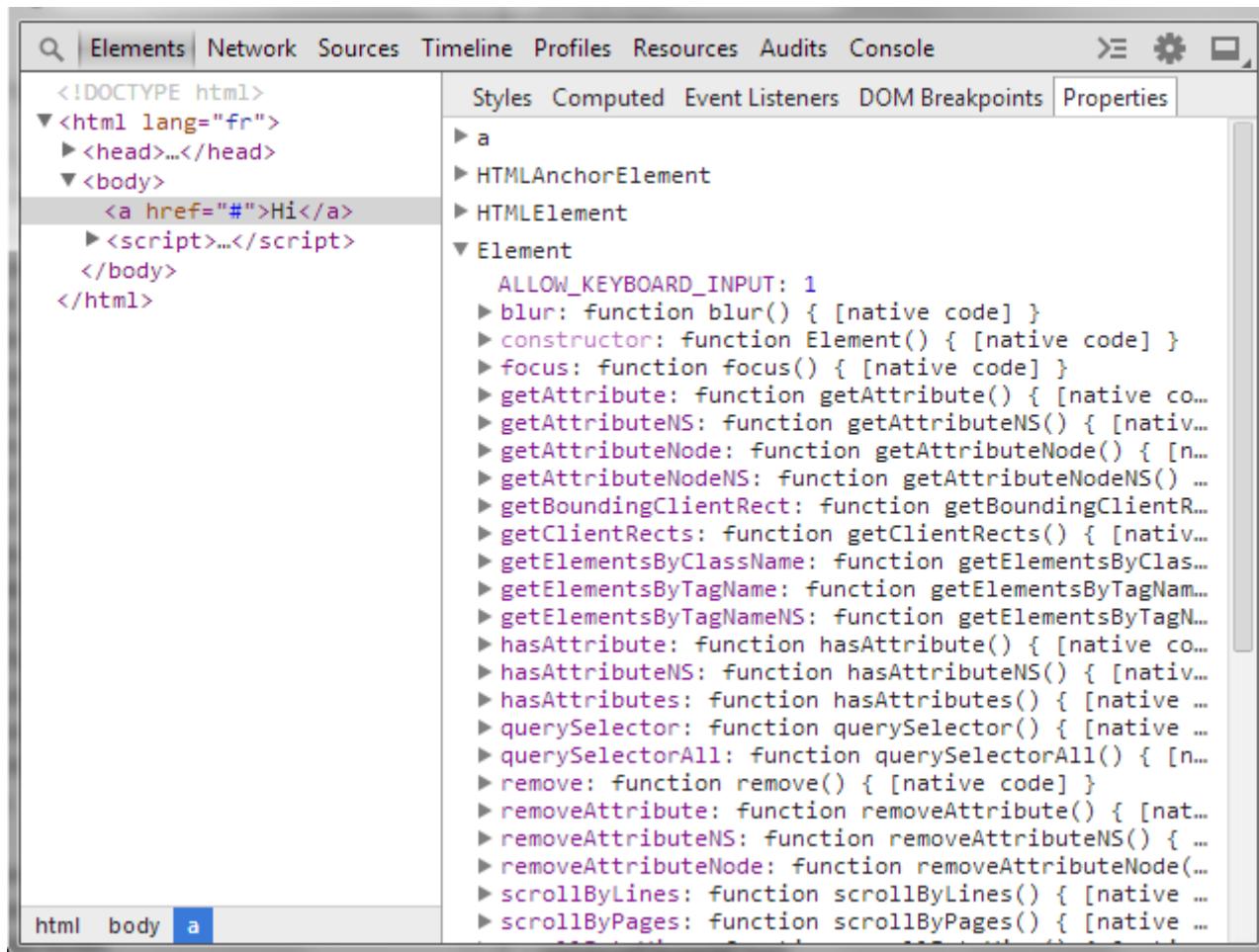
The screenshot shows the Google Chrome DevTools Console tab. It displays a list of properties and their values for the global `document` object. The properties listed are: `doctype`, `nodeName`, `nodeType`, `querySelector`, `createDocumentFragment`, and `querySelector('a')`. Each property is followed by its value and the line number where it was defined or last modified, which is `test3.html:8` for all properties except `nodeType` which is `9`.

Property	Value	Line Number
<code>document.doctype</code>	<code>> <!DOCTYPE html></code>	<code>test3.html:8</code>
<code>document.doctype.nodeName</code>	<code>html</code>	<code>test3.html:9</code>
<code>document.doctype.nodeType</code>	<code>10</code>	<code>test3.html:10</code>
<code>document.nodeName</code>	<code>#document</code>	<code>test3.html:13</code>
<code>document.nodeType</code>	<code>9</code>	<code>test3.html:14</code>
<code>document.createDocumentFragment().nodeName</code>	<code>#document-fragment</code>	<code>test3.html:18</code>
<code>document.createDocumentFragment().nodeType</code>	<code>11</code>	<code>test3.html:20</code>
<code>document.querySelector('a').nodeName</code>	<code>A</code>	<code>test3.html:24</code>
<code>document.querySelector('a').nodeType</code>	<code>1</code>	<code>test3.html:26</code>
<code>document.querySelector('a').firstChild.nodeName</code>	<code>#text</code>	<code>test3.html:30</code>
<code>document.querySelector('a').firstChild.nodeType</code>	<code>3</code>	<code>test3.html:32</code>

Le meilleur moyen de déterminer si un nœud est d'un certain type est de vérifier simplement sa propriété `nodeType`, comme on l'a fait dans l'exemple ci-dessus avec la balise `<a>` et la ligne de code suivante :

```
console.log("document.querySelector('a').firstChild.nodeType => " ,  
document.querySelector('a').firstChild.nodeType);
```

Etre capable de déterminer le type de noeud d'un élément HTML est intéressant car cela permet de déterminer précisément de quelles propriétés et méthodes on dispose pour cet élément. Mais on peut aussi le savoir très rapidement en s'appuyant sur les outils de développement du navigateur, comme par exemple ceux de Google Chrome :



4.1.7 Récupérer la valeur d'un noeud

La propriété "nodeValue" renvoie "null" la plupart du temps, sauf pour les noeuds de type "Text" et "Comment".

On réservera donc l'usage de cette méthode aux cas où on souhaite récupérer le contenu textuel encapsulé par un élément, comme par exemple le texte se trouvant entre les balises `<a>` et ``, ou encore `<td>` et `</td>`.

L'exemple de code ci-dessous permet de sa familiariser avec l'utilisation de cette propriété, et de voir dans quels cas son utilisation est pertinente :

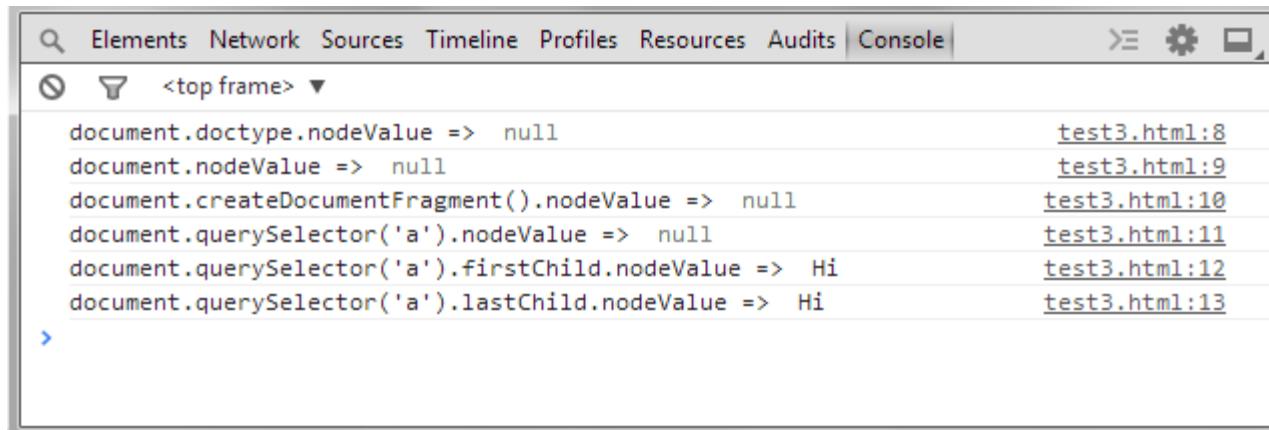
```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
```

```
<body>
  <a href="#">Hi</a>

  <script>
    console.log('document.doctype.nodeValue => ' ,
document.doctype.nodeValue);
    console.log('document.nodeValue => ' , document.nodeValue);
    console.log('document.createDocumentFragment().nodeValue => ' ,
document.createDocumentFragment().nodeValue);
    console.log("document.querySelector('a').nodeValue => " ,
document.querySelector('a').nodeValue);
    console.log("document.querySelector('a').firstChild.nodeValue => " ,
document.querySelector('a').firstChild.nodeValue);
    console.log("document.querySelector('a').lastChild.nodeValue => " ,
document.querySelector('a').lastChild.nodeValue);
  </script>

</body>
</html>
```

Consultation de la log du navigateur :



```
Elements Network Sources Timeline Profiles Resources Audits | Console
<top frame> ▾
document.doctype.nodeValue => null test3.html:8
document.nodeValue => null test3.html:9
document.createDocumentFragment().nodeValue => null test3.html:10
document.querySelector('a').nodeValue => null test3.html:11
document.querySelector('a').firstChild.nodeValue => Hi test3.html:12
document.querySelector('a').lastChild.nodeValue => Hi test3.html:13
```

Je vous invite à faire une petite expérience : ajoutez la ligne ci-dessous sur la ligne de commande de la console (ou dans la partie "<script>" de la page, et observez ce qui se passe :

```
document.querySelector('a').lastChild.nodeValue = 'hello!!' ;
```

4.1.8 Insertion de HTML avec ou sans parsing

Les propriétés innerHTML, outerHTML, textContent, et la méthode insertAdjacentHTML() fournissent des outils très pratiques pour insérer des éléments et du texte dans une page HTML (et donc dans le DOM sous-jacent). Voici quelques exemples d'utilisations :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>

<div id="A"></div>
<span id="B"></span>
<div id="C"></div>
<div id="D"></div>
<div id="E"></div>
<script>
    /* Sélectionne l'élément ayant pour ID "A" et modifie son contenu
       textuel en insérant une balise <strong> avec un peu de texte */
    document.getElementById('A').innerHTML = '<strong>Salut !!</strong>';
    /* Sélectionne l'élément <span> ayant pour ID "B" et le remplace par un
       nouvel élément de type <div> */
    document.getElementById('B').outerHTML = '<div id="B" class="new">Quoi
de neuf ?</div>';
    /* Sélectionne l'élément ayant pour ID "C" et y insère un nouvel noeud
       de type "text node" */
    document.getElementById('C').textContent = 'frangin';
    /* Propriétés NON standard innerText pour insérer du contenu textuel
       dans la <div> ayant pour ID "D" */
    document.getElementById('D').innerText = 'Attrape ça !!!';
    /* Propriétés NON standard outerText pour insérer du contenu textuel en
       remplacement de la <div> ayant pour ID "E" */
    document.getElementById('E').outerText = 'j\'adore Javascript !';
    /* Prenon une photographie du contenu HTML de <body> après le passage du
       code Javascript */
    console.log(document.body.innerHTML);
    /* logs
    <div id="A"><strong>Salut !!</strong></div>
    <div id="B" class="new">Quoi de neuf ?</div>
    <div id="C">frangin</div>
    <div id="D">Attrape ça !!!</div>
    j'adore Javascript !
    */
</script>

</body>
</html>
```

La méthode insertAdjacentHTML() est très pratique pour insérer des éléments avant ou après d'autres éléments.

Démonstration par l'exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>

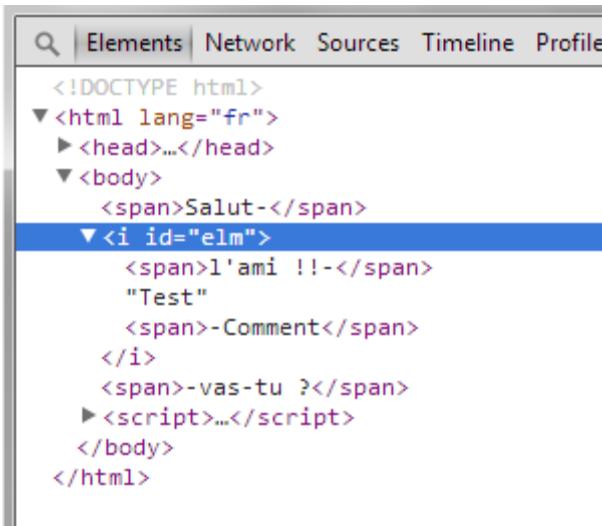
    <i id="elm">Test</i>
    <script>
        var elm = document.getElementById('elm');
        elm.insertAdjacentHTML('beforebegin', '<span>Salut-</span>');
        elm.insertAdjacentHTML('afterbegin', '<span>l\'ami !!-</span>');
        elm.insertAdjacentHTML('beforeend', '<span>-Comment</span>');
        elm.insertAdjacentHTML('afterend', '<span>-vas-tu ?</span>');

        console.log(document.body.innerHTML);

    /* logs
    <span>Salut-</span><i id="elm"><span>l'ami !!-</span>Test<span>-Comment</span></i><span>-vas-tu ?</span>
    */
    </script>

</body>
</html>
```

La représentation du DOM au travers des outils de développement aide à comprendre ce qui s'est passé :



Points à noter :

La propriété innerHTML a pour effet de "parser" le code transmis pour le transformer en éléments HTML, et donc en nouveaux noeuds à l'intérieur du DOM. Ce mécanisme de parsing est puissant mais également coûteux en ressources. Si l'objectif est d'insérer du texte ne nécessitant pas la création de nouveaux éléments dans le DOM, alors on aura tout intérêt à utiliser la propriété textContent.

La méthode document.write () peut également être utilisée pour créer des noeuds et les ajouter au DOM. Cependant , elle est généralement peu utilisée, sauf cas très particulier, car elle a pour effet néfaste de bloquer l'analyse du code HTML dans le document en cours de chargement.

Les options "beforebegin" et "afterend" de la méthode insertAdjacentHTML() ne fonctionneront que si le noeud concerné est attaché à un élément parent à l'intérieur de l'arbre DOM.

A noter que le support pour outerHTML n'était pas disponible nativement dans Firefox jusqu'à la version 11 (on est actuellement à la version 26, donc...).

A noter également que les méthodes InsertAdjacentElement () et insertAdjacentText () sont disponibles sur la plupart des navigateurs modernes, sauf Firefox.

4.1.9 Extraire des éléments de l'arbre DOM

Puisque l'on peut utiliser les propriétés innerHTML, outerHTML et textContent pour modifier ou insérer des éléments, on peut aussi les utiliser pour récupérer le contenu d'éléments HTML sélectionnés :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>

    <div id="A">
        <i>Salut</i>
    </div>
    <div id="B">
        l'ami <strong> !</strong>
    </div>
    <script>
        console.log(document.getElementById('A').innerHTML);
        /* logs '<i>Salut</i>' */

        console.log(document.getElementById('A').outerHTML);
        /* logs "<div id='A'><i>Salut</i></div>" */

        console.log(document.getElementById('B').innerHTML);
        /* logs "l'ami <strong> !</strong>" */

        console.log(document.getElementById('B').textContent);
        /* logs 'l'ami !' */

        //propriétés NON standard
        console.log(document.getElementById('B').innerText); //logs 'l'ami !
        console.log(document.getElementById('B').outerText); //logs 'l'ami !'
    </script>

</body>
</html>
```

Note

Les propriétés textContent, innerText et outerText renvoient tous les nœuds de texte contenus dans le nœud sélectionné.

Ainsi, à titre d'exemple (notez que ce n'est pas une bonne idée dans la pratique), document.body.textContent renvoie tous les nœuds de texte contenus dans le corps de la page.

4.1.10 Ajouter des noeuds textuels dans le DOM

Dans le même temps où le navigateur "parse" un document HTML, il construit l'arbre DOM correspondant.

Il est possible d'intervenir sur cet arbre par la suite, pour y ajouter des éléments.

Les principales méthodes pour ajouter des éléments sont :

- createElement()
- createTextNode()

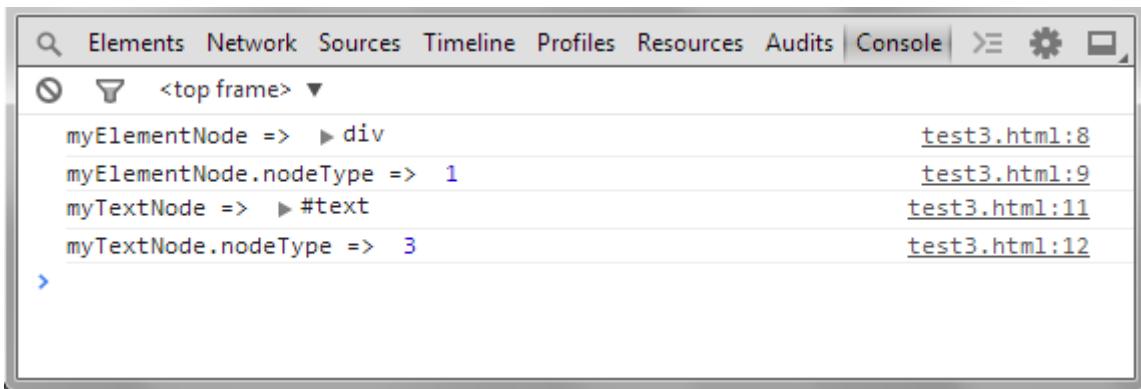
D'autres méthodes sont disponibles mais peu utilisées, comme createAttribute() et createComment(). La méthode createAttribute() est dépréciée et on recommande donc de ne plus l'utiliser. La méthode createComment() est peu utilisée, mais elle n'est pas dépréciée et vous pouvez l'utiliser si son usage vous semble pertinent.

Exemple de code pour la création de 2 éléments dans le DOM :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>

    <script>
        var myElementNode = document.createElement('div');
        console.log('myElementNode => ', myElementNode);
        console.log('myElementNode.nodeType => ', myElementNode.nodeType);
        var myTextNode = document.createTextNode('Bonjour, comment allez-vous
?');
        console.log('myTextNode => ', myTextNode);
        console.log('myTextNode.nodeType => ', myTextNode.nodeType);
    </script>

</body>
</html>
```



Une chose vous a peut être sauté aux yeux, rien n'apparaît pour l'instant dans la page affichée par le navigateur. C'est normal, les 2 éléments sont pour l'instant bien créés quelque part dans la mémoire du navigateur, mais ils ne sont pas encore rattachés à l'arbre DOM. Ils n'ont donc aucune raison d'apparaître. Comment les rattacher ? C'est ce que nous allons voir dans le chapitre suivant.

4.1.11 Ajout de noeuds avec appendChild() et insertBefore()

Pour insérer notre nouvelle "div" dans le DOM, il nous faut l'indiquer explicitement au navigateur. La méthode la plus fréquemment utilisée pour cela est la méthode appendChild().

Il nous faut tout d'abord attacher l'élément "myTextNode" à l'élément "myElementNode". Puis il nous faut lier ensemble l'élément "body" et l'élément "myElementNode". C'est ce que font les 2 lignes utilisant la méthode appendChild() dans l'exemple ci-dessous :

```

<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>

<script>
    var myElementNode = document.createElement('div');
    console.log('myElementNode => ', myElementNode);
    console.log('myElementNode.nodeType => ', myElementNode.nodeType);
    var myTextNode = document.createTextNode('Bonjour, comment allez-vous
?');
    console.log('myTextNode => ', myTextNode);
    console.log('myTextNode.nodeType => ', myTextNode.nodeType);

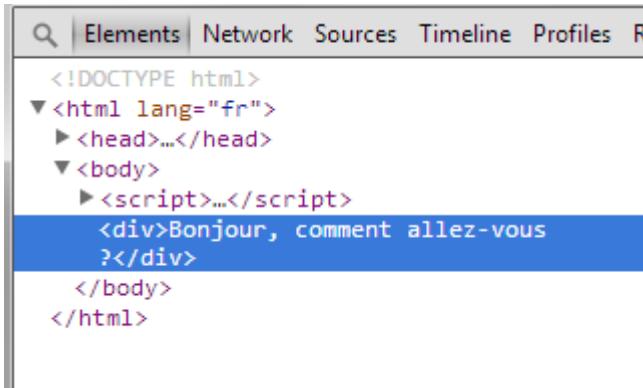
    myElementNode.appendChild(myTextNode) ; /* lier ensemble myTextNode et
myElementnode */
    document.body.appendChild(myElementNode) ; /* lier ensemble

```

```
myElementnode et body */
</script>

</body>
</html>
```

Il est intéressant de noter où la nouvelle "div" a été insérée dans la page par le navigateur.



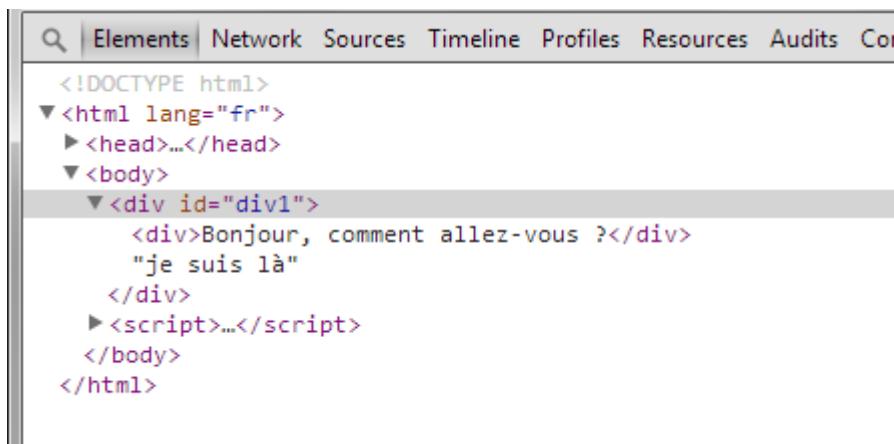
A noter que l'on peut insérer la nouvelle balise "div" plus précisément à l'intérieur de la page au moyen de la méthode insertBefore(), comme dans l'exemple suivant :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
  <div id="div1">je suis là</div>
  <script>
    var myElementNode = document.createElement('div');
    console.log('myElementNode => ', myElementNode);
    console.log('myElementNode.nodeType => ', myElementNode.nodeType);
    var myTextNode = document.createTextNode('Bonjour, comment allez-vous
?');
    console.log('myTextNode => ', myTextNode);
    console.log('myTextNode.nodeType => ', myTextNode.nodeType);

    myElementNode.appendChild(myTextNode) ;
    var selection = document.getElementById('div1');
    selection.insertBefore(myElementNode, selection.firstChild) ;
  </script>

</body>
</html>
```

D'un point de vue de l'arbre DOM, cela donne :



Autre exemple de manipulation et d'insertion dans le DOM : on souhaite ajouter un élément au tout début d'une liste HTML non ordonnée ().

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <ul>
        <li>2</li>
        <li>3</li>
    </ul>
    <script>
        // création d'un élément de type textNode
        var text1 = document.createTextNode('1');
        // création d'un élément de type "li"
        var li = document.createElement('li');
        // rattachement de l'élément de type "textNode" à l'élément de type "li"
        li.appendChild(text1);
        // sélection de l'élément "ul" dans le document
        var ul = document.querySelector('ul');
        /*
            ajout de l'élément "li" devant le premier enfant de l'élément "ul"
            sélectionné
        */
        ul.insertBefore(li, ul.firstChild);
        console.log(document.body.innerHTML);
        /*logs
        <ul>
            <li>1</li>
            <li>2</li>
            <li>3</li>
        </ul>
        */
    </script>

</body>
</html>
```

La méthode insertBefore() nécessite 2 paramètres :

- le noeud à insérer
- une référence au noeud "cible" devant lequel on souhaiter effectuer l'insertion

4.1.12 Supprimer et remplacer des noeuds dans le DOM

La suppression de noeuds se fait en plusieurs étapes :

1. il faut d'abord sélectionner l'élément à supprimer
2. il faut ensuite accéder à l'élément "parent" de l'élément à supprimer, en utilisant la propriété "parentNode"
3. à partir du noeud "parent", on va pouvoir faire appel à la méthode removeChild(), en lui passant la référence du noeud à supprimer.

Démonstration par l'exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <div id="A">Salut</div>
    <div id="B">l'ami !</div>
    <script>
        // on veut supprimer l'élément ayant pour ID "A"
        var divA = document.getElementById('A');
        divA.parentNode.removeChild(divA);
        // on veut supprimer le premier élément situé dans l'élément ayant pour
ID "B"
        var divB = document.getElementById('B').firstChild;
        divB.parentNode.removeChild(divB);
        // il n'y a plus grand chose à voir dans la page... et dans le DOM ?
        console.log(document.body.innerHTML); // logs "<div id="B"></div>",
plus un peu de javascript
    </script>
</body>
</html>
```

Le remplacement d'un élément fonctionne sur le même principe, si ce n'est qu'on utilise la méthode replaceChild() :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
```

```

<body>
  <div id="A">Donald</div>
  <div id="B">Dingo</div>
  <script>
    // remplacement d'un élément de type noeud
    var divA = document.getElementById('A');
    var newSpan = document.createElement('span');
    newSpan.textContent = 'Mickey';
    divA.parentNode.replaceChild(newSpan, divA);

    // remplacement d'un élément de type texte
    var divB = document.getElementById('B').firstChild;
    var newText = document.createTextNode('Minnie');
    divB.parentNode.replaceChild(newText, divB);

    // log du DOM mis à jour, qui devrait contenir : <span>Mickey</span> <div
    id="B">Minnie</div>
    console.log(document.body.innerHTML);
  </script>
</body>
</html>

```

Notes

Les méthodes `replaceChild()` et `removeChild()` renvoient toutes deux une référence à l'élément remplacé ou supprimé, qui est toujours virtuellement présent, même s'il n'apparaît dans le document en cours d'affichage. Il est donc possible de récupérer la référence à cet élément (remplacé ou supprimé) dans une variable pour le manipuler, ou le faire réapparaître à un autre endroit de la page par exemple.

4.1.13 Cloner des éléments du DOM

En utilisant la méthode `cloneNode()`, il est possible de dupliquer un noeud seul, ou un noeud et tous ses noeuds "enfants".

Dans l'exemple ci-dessous, seul l'élément "ul" est cloné :

```

<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
  <ul>
    <li>Hi</li>
    <li>there</li>
  </ul>
  <script>
    var cloneUL = document.querySelector('ul').cloneNode();

```

```
        console.log(cloneUL.constructor); // logs function HTMLULListElement() {
[native code] }
        console.log(cloneUL.innerHTML); // logs (une chaîne vide) car seul
l'élément "ul" a été cloné

    </script>
</body>
</html>
```

Mais en passant un paramètre fixé à "true" à la méthode cloneNode(), on obtient un clone contenant l'intégralité de la liste :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <ul>
        <li>Hi</li>
        <li>there</li>
    </ul>
    <script>
        var cloneUL = document.querySelector('ul').cloneNode(true);
        console.log(cloneUL.constructor); // function HTMLULListElement() {
[native code] }
        console.log(cloneUL.innerHTML); // logs <li>Hi</li><li>there</li>
    </script>
</body>
</html>
```

4.1.14 Parcourir le DOM

A partir d'un noeud de référence (obtenu par exemple avec : `document.querySelector('ul')`), il est possible de parcourir le DOM en utilisant les propriétés suivantes :

- `parentNode`
- `firstChild`
- `lastChild`
- `nextSibling`
- `previousSibling`

```
<html lang="fr">
<meta charset="utf-8">
<body>
    <ul>
        <!-- comment -->
        <li id="A">ligne 1</li>
        <li id="B">ligne 2</li>
        <!-- comment -->
    </ul>
    <script>
        //cache selection of the ul
        var ul = document.querySelector('ul');

        //What is the parentNode of the ul?
        console.log('ul.parentNode.nodeName => ', ul.parentNode.nodeName);
        //logs BODY
        console.log('ul.parentNode.innerText => ', ul.parentNode.innerText);
        //logs ul.parentNode.innerText => ligne 1 ligne 2

        //What is the first child of the ul?
        console.log('ul.firstChild.nodeName => ', ul.firstChild.nodeName);
        //logs #text
        console.log('ul.firstChild.innerText => ', ul.firstChild.innerText);
        //logs undefined

        //What is the last child of the ul?
        console.log('ul.lastChild.nodeName => ', ul.lastChild.nodeName);
        /* logs text not comment, because there is a line break */

        //What is the nextSibling of the first li?
        console.log("ul.querySelector('#A').nextSibling.innerHTML => ",
            ul.querySelector('#A').nextSibling.innerHTML); //logs text

        //What is the previousSibling of the last li?
        console.log("ul.querySelector('#B').previousSibling.nodeName => ",
            ul.querySelector('#B').previousSibling.nodeName); //logs text
```

```
        console.log("ul.querySelector('#B').previousSibling.innerHTML => ",  
        ul.querySelector('#B').previousSibling.innerHTML);  
        //logs undefined  
  
    </script>  
</body>  
</html>
```

Si vous êtes déjà familiarisé avec le DOM, vous ne serez peut être pas surpris par le fait que parcourir le DOM ne se résume pas à parcourir les seuls éléments visibles. En effet, dans les exemples précédents, on a été fréquemment en contact avec des noeuds de types "texte" et "commentaires", qui nous ont renvoyé des informations généralement peu intéressantes.

Dans l'exemple ci-dessus, le fait d'écrire `ul.querySelector('#A').nextSibling` nous positionne sur le caractère "saut de ligne" que l'on voit implicitement dans le code source de la page, et non pas sur l'élément de type "li" qui suit immédiatement sur lequel on est en train de pointer (en l'occurrence celui qui a pour ID "A").

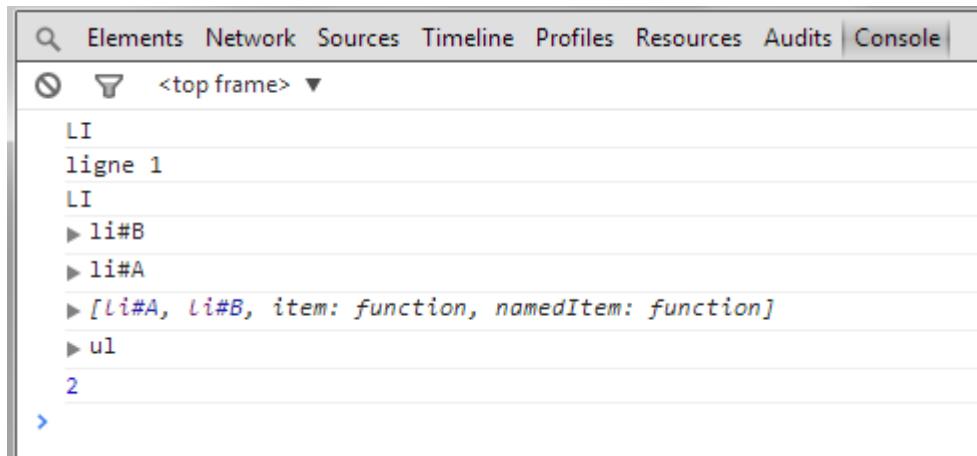
Si l'on souhaite parcourir le DOM en ignorant ces informations que l'on pourrait juger - dans certains cas - comme "parasite", alors il faut recourir aux méthodes suivantes :

- `firstElementChild`
- `lastElementChild`
- `nextElementChild`
- `previousElementChild`
- `children`
- `parentElement`
- `nextElementSibling`
- `previousElementSibling`

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <ul>
        <!-- comment -->
        <li id="A">ligne 1</li>
        <li id="B">ligne 2</li>
        <!-- comment -->
    </ul>
    <script>
        //cache selection of the ul
        var ul = document.querySelector('ul');
        //What is the first child of the ul?
        console.log(ul.firstElementChild.nodeName); //logs li
        //What is the last child of the ul?
        console.log(ul.lastElementChild.nodeName); //logs li
        //What is the nextSibling of the first li?
        console.log(ul.querySelector('#A').nextElementSibling.nodeName); //logs
        li
        //What is the previousSibling of the last li?
        console.log(ul.querySelector('#B').previousElementSibling.nodeName);
        //logs li
        //What are the element only child nodes of the ul?
        console.log(ul.children); //HTMLCollection, all child nodes including
        text nodes
        //What is the parent element of the first li?
        console.log(ul.firstElementChild.parentElement); //logs ul
    </script>
</body>
</html>
```

On obtient cette fois une liste d'éléments beaucoup plus intéressante :



4.1.15 Vérifier la position d'un noeud

Il est quelquefois nécessaire de savoir si un noeud est bien un enfant d'un autre noeud.

On peut le vérifier facilement au moyen de la méthode `contains()`, comme dans l'exemple suivant, dans lequel on vérifie que l'élément "html" contient bien l'élément "body" :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
  <script>
    // est-ce que <body> se trouve à l'intérieur de <html> ?
    var inside =
document.querySelector('html').contains(document.querySelector('body'));
    console.log(inside); //logs true
  </script>
</body>
</html>
```

4.2 Le noeud "document"

4.2.1 Introduction

Dans la pratique, le développeur Javascript manipule beaucoup plus souvent les enfants de l'objet "document" que l'objet "document" lui-même.

Nous allons néanmoins présenter brièvement quelques unes des caractéristiques de l'objet "document".

Plusieurs propriétés et méthodes sont accessibles au travers de l'objet "document" :

- doctype
- documentElement
- implementation.*
- activeElement
- body

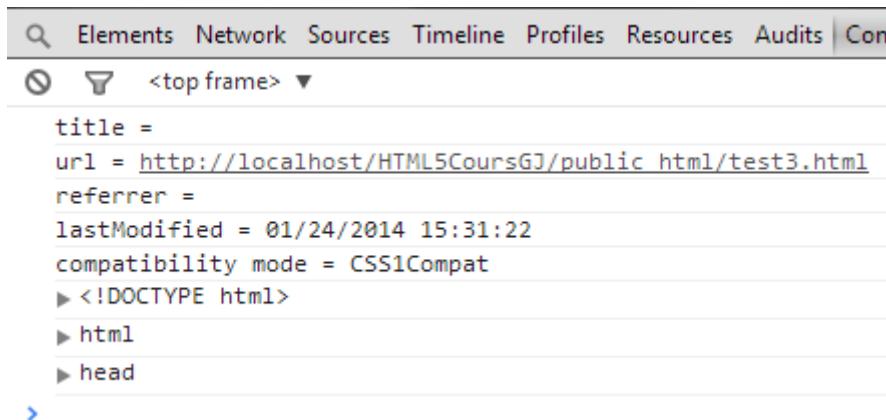
- head
- title
- lastModified
- referrer
- URL
- defaultView
- compatMode
- ownerDocument
- hasFocus()

Quelques exemples d'utilisation :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <script>
        var d = document;
        console.log('title = ' + d.title);
        console.log('url = ' + d.URL);
        console.log('referrer = ' + d.referrer);
        console.log('lastModified = ' + d.lastModified);

        //log soit BackCompat (Quirks Mode) ou CSS1Compat (Strict Mode)
        console.log('compatibility mode = ' + d.compatMode);

        console.log(document.doctype);
        console.log(document.documentElement);
        console.log(document.head);
    </script>
</body>
</html>
```



The screenshot shows the Chrome DevTools Elements tab with the search bar empty. Below it, the page source is displayed. The output of the script is shown in the console:

```
<top frame>
title =
url = http://localhost/HTML5CoursGJ/public_html/test3.html
referrer =
lastModified = 01/24/2014 15:31:22
compatibility mode = CSS1Compat
▶ <!DOCTYPE html>
▶ html
▶ head
```

4.2.2 Déterminer où est le focus

En utilisant la propriété document.activeElement, on peut savoir très rapidement sur quel élément de la page le focus est placé.

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <textarea id="field01"></textarea>
    <script>
        //set focus to <textarea>
        document.querySelector('textarea').focus();

        //get reference to element that is focused/active in the document
        console.log(document.activeElement); //logs <textarea
        id="field01"></textarea>
    </script>
</body>
</html>
```

4.3 Les noeuds du DOM

4.3.1 Element Nodes

4.3.1.1 HTML*Element - Introduction

Chaque élément d'un document HTML est rattaché à un constructeur Javascript qui instancie l'élément comme un nœud dans une arborescence DOM. Par exemple, un élément `<a>` est créé comme un nœud DOM défini par la fonction "constructeur" `HTMLAnchorElement()`. Dans l'exemple suivant, on vérifie que la balise "a" est créée à partir de la fonction `HTMLAnchorElement()` en faisant appel à la propriété "constructor", après avoir sélectionné l'élément "a" via la méthode `querySelector()`.

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <a></a>
    <script>
        console.log(document.querySelector('a').constructor); /* logs function
        HTMLAnchorElement() { [native code] } */
    </script>

```

```
</body>
</html>
```

La liste suivante donne un aperçu des fonctions les plus fréquemment utilisées par le navigateur, pour créer chacun des éléments du DOM lié à une page HTML :

HTMLHtmlElement	HTMLParagraphElement
HTMLHeadElement	HTMLHeadingElement
HTMLLinkElement	HTMLQuoteElement
HTMLTitleElement	HTMLPreElement
HTMLMetaElement	HTMLBRElement
HTMLBaseElement	HTMLBaseFontElement
HTMLIndexElement	HTMLFontElement
HTMLStyleElement	HTMLHRElement
HTMLBodyElement	HTMLModElement
HTMLFormElement	HTMLAnchorElement
HTMLSelectElement	HTMLImageElement
HTMLOptGroupElement	HTMLObjectElement
HTMLOptionElement	HTMLParamElement
HTMLInputElement	HTMLAppletElement
HTMLTextAreaElement	HTMLMapElement
HTMLButtonElement	HTMLAreaElement
HTMLLabelElement	HTMLScriptElement
HTMLFieldSetElement	HTMLTableElement
HTMLLegendElement	HTMLTableCaptionElement
HTMLULListElement	HTMLTableColElement
HTMLOLListElement	HTMLTableSectionElement
HTMLDLListElement	HTMLTableRowElement
HTMLDirectoryElement	HTMLTableCellElement
HTMLMenuElement	HTMLFrameSetElement
HTMLLIElement	HTMLFrameElement
HTMLDivElement	HTMLIFrameElement

Il est important de garder à l'esprit que chaque élément HTML défini par l'une des fonctions précédentes hérite des propriétés et méthodes de : `HTMLElement`, `Element`, `Node`, et `Object`.

Donc, quand vous insérez dans le DOM une balise `<a>` avec la méthode suivante :

```
document.createElement('div');
```

... vous faites appel sans le savoir au constructeur `HTMLAnchorElement` qui va vous permettre d'hériter de toutes les propriétés et méthodes avec lesquelles vous allez travailler par la suite (par exemple : `getElementById()`, `setAttribute()`, `querySelector()`, etc...).

4.3.1.2 HTML*Element Object - Propriété et méthodes

Pour savoir précisément de quelles méthodes et propriétés vous disposez, plutôt que de passer beaucoup de temps à étudier la documentation du W3C (très intéressante mais particulièrement verbeuse), il est préférable de faire confiance au navigateur et de lui demander de vous fournir la liste des propriétés et méthodes disponibles :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <a href="#">Hi</a>
    <script>
        var anchor = document.querySelector('a');

        // Propriétés appartenant à l'élément sélectionné
        console.log(Object.keys(anchor).sort());

        // Propriétés appartenant à l'élément sélectionné et propriétés héritées
        var documentPropertiesIncludeInherited = [];
        for ( var p in document) {
            documentPropertiesIncludeInherited.push(p);
        }
        console.log(documentPropertiesIncludeInherited.sort());

        // Propriétés héritées seulement :
        var documentPropertiesOnlyInherited = [];
        for ( var p in document) {
            if (!document.hasOwnProperty(p)) {
                documentPropertiesOnlyInherited.push(p);
            }
        }
        console.log(documentPropertiesOnlyInherited.sort());

    </script>
</body>
</html>
```

Jetez un coup d'oeil du côté de la console pour voir ce que cela donne... Oui, je sais, la première fois ça fait un choc :

The screenshot shows the Chrome DevTools console tab with the title '<top frame>' selected. The console output displays a large list of properties and methods for the `Element` object, which is indicated by the first item in the list being preceded by a right-pointing arrow.

```

▶ ["accessKey", "attributes", "baseURI", "charset", "childElementCount", "childNodes", "childTextEditable", "coords", "dataset", "dir", "download", "draggable", "firstChild", "firstElementText", "isContentEditable", "lang", "lastChild", "lastElementChild", "LocalName", "name", "setHeight", "offsetLeft", "offsetParent", "offsetTop", "offsetWidth", "onabort", "onbeforeonchange", "onclick", "onclose", "oncontextmenu", "oncopy", "oncuechange", "oncut", "ondblclick", "ondurationchange", "onemptied", "onended", "onerror", "onfocus", "oninput", "oninvalid", "onstart", "onmousedown", "onmouseenter", "onmouseleave", "onmousemove", "onmouseout", "onmouseout", "onmou", "onratechange", "onreset", "onscroll"]...
```

```

▶ ["ATTRIBUTE_NODE", "CDATA_SECTION_NODE", "COMMENT_NODE", "DOCUMENT_FRAGMENT_NODE", "DOCUMENT_NCONNECTED", "DOCUMENT_POSITION_FOLLOWING", "DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC", "DRENCE_NODE", "NOTATION_NODE", "PROCESSING_INSTRUCTION_NODE", "TEXT_NODE", "URL", "activeElei", "baseURI", "bgColor", "body", "captureEvents", "caretRangeFromPoint", "characterSet", "reDocumentPosition", "compatMode", "contains", "cookie", "createAttribute", "createAttribut", "createElementNS", "createEvent", "createExpression", "createNSResolver", "createNodeIterator", "ipt", "defaultCharset", "defaultView", "designMode", "dir", "dispatchEvent", "doctype", "do d", "fgColor", "firstChild", "firstElementChild", "forms", "getCSSCanvasContext", "getEleme", "TagNameNS", "getOverrideStyle", "getSelection", "hasChildNodes", "hasFocus", "head", "image EqualNode", "isSameNode", "isSupported", "lastChild", "lastElementChild", "lastModified"]...
```

```

["ATTRIBUTE_NODE", "CDATA_SECTION_NODE", "COMMENT_NODE", "DOCUMENT_FRAGMENT_NODE", "DOCUMENT_NNNECTED", "DOCUMENT_POSITION_FOLLOWING", "DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC", "DOCUME NODE", "NOTATION_NODE", "PROCESSING_INSTRUCTION_NODE", "TEXT_NODE", "addEventListener", "adop compareDocumentPosition", "contains", "createAttribute", "createAttributeNS", "createCDATASec ent", "createExpression", "createNSResolver", "createNodeIterator", "createProcessingInstruct "evaluate", "execCommand", "getCSSCanvasContext", "getElementById", "getElementsByClassName", tSelection", "hasChildNodes", "hasFocus", "importNode", "insertBefore", "isDefaultNamespace", "open", "queryCommandEnabled", "queryCommandIndeterm", "queryCommandState", "queryCommandSupp ", "removeEventListener", "replaceChild", "webkitCancelFullScreen", "webkitExitFullscreen", "...
```

Si beaucoup de propriétés et méthodes sont disponibles, en règle générale on en utilisera un petit nombre, à commencer par celles ci-dessous :

- createElement()
- tagName
- children
- getAttribute()
- setAttribute()
- hasAttribute()
- removeAttribute()
- classList()
- dataset
- attributes

4.3.1.3 Récupérer la liste des attributs d'un élément

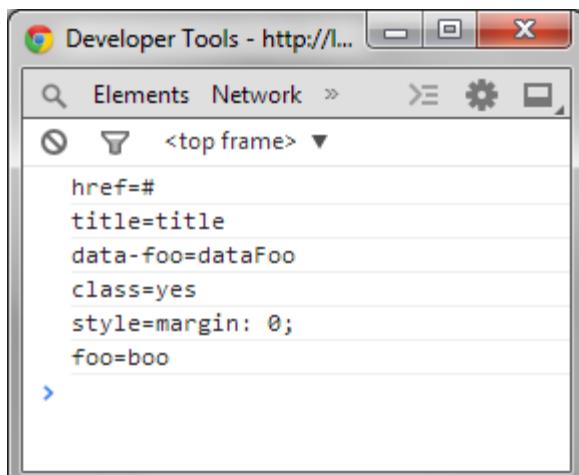
Pour rappel, on a vu dans le chapitre présentant en détail le fonctionnement du DOM, comment créer un élément (par exemple une balise `<a>`).

On a vu également qu'il était possible de récupérer le nom du tag et le nom du noeud au moyen d'instructions telles que celles-ci :

```
console.log(document.querySelector('a').tagName);
console.log(document.querySelector('a').nodeName);
```

En utilisant maintenant les propriétés "attributs", nous pouvons récupérer la liste des noeuds de type "Attr" qui sont rattachés à un élément donné. La liste récupérée est désignée sous le nom de `NamedNodeMap`.

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <a href="#" title="title" data-foo="dataFoo" class="yes" style="margin: 0;" foo="boo"></a>
    <script>
        var atts = document.querySelector('a').attributes;
        for ( var i = 0, len = atts.length ; i < len; i++) {
            console.log(atts[i].nodeName + '=' + atts[i].nodeValue);
        }
    </script>
</body>
</html>
```



4.3.1.4 Ajouter, supprimer, récupérer les valeurs d'attributs

La manière la plus simple et aussi la plus efficace pour définir, récupérer ou supprimer des attributs consiste à utiliser les méthodes `getAttribute()`, `setAttribute()` et `removeAttribute()`.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <a href="#" title="title" data-foo="dataFoo" style="margin: 0;">#link</a>
    <script>
        var atts = document.querySelector('a');

        // suppression des attributs
        atts.removeAttribute('href');
        atts.removeAttribute('title');
        atts.removeAttribute('style');
        atts.removeAttribute('data-foo');
        atts.removeAttribute('class');
        atts.removeAttribute('foo'); //custom attribute
        atts.removeAttribute('hidden'); //boolean attribute

        // (re)création des attributs
        atts.setAttribute('href', '#');
        atts.setAttribute('title', 'Titre');
        atts.setAttribute('style', 'margin:0;');
        atts.setAttribute('data-foo', 'dataFoo');
        atts.setAttribute('class', 'yes');
        atts.setAttribute('foo', 'boo');
        atts.setAttribute('hidden', 'hidden'); /* boolean attribute
                                                requires sending the attribute as the value too */

        // récupération de la valeur des attributs
        console.log(atts.getAttribute('href'));
        console.log(atts.getAttribute('title'));
        console.log(atts.getAttribute('style'));
        console.log(atts.getAttribute('data-foo'));
        console.log(atts.getAttribute('class'));
        console.log(atts.getAttribute('foo'));
        console.log(atts.getAttribute('hidden'));

    </script>
</body>
</html>
```

Points à noter :

Certains vieux navigateurs (comme IE 7, et peut être IE 8, mais c'est à vérifier) ne supportaient pas l'utilisation des méthodes `setAttribute()` et `getAttribute()` sur les paramètres "style" et "class". Pour fonctionner avec ces navigateurs, on devrait remplacer les appels à la méthode `getAttribute()` par :

```
//get attributes
console.log(att.getAttribute("href"));
console.log(att.getAttribute("title"));
console.log(att.getAttribute("style"));
console.log(att.getAttribute("dataset.dataFoo"));
console.log(att.getAttribute("className"));
console.log(att.getAttribute("foo")); // renvoie "undefined"
console.log(att.getAttribute("hidden"));
```

Mais cette technique ne fonctionne pas avec les attributs `data-*` (cf. chapitre "Manipuler les attributs `data-*`" plus loin dans ce document).

De plus, l'attribut "foo" est inaccessible, son invocation renvoie la valeur "undefined". On notera que l'attribut "class" doit être remplacé par le mot clé "className" pour pouvoir récupérer la valeur associée à l'attribut "class".

A moins d'être contraint au maintien d'une compatibilité avec de vieux navigateurs, on préfèrera utiliser les méthodes `getAttribute()` et `setAttribute()`, qui offrent une meilleure lisibilité, et permettent de travailler avec tous les attributs.

4.3.1.5 Vérifier la présence d'un attribut

Le meilleur moyen pour déterminer si un élément du DOM a un attribut particulier consiste à utiliser la méthode `hasAttribute()`.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <a href="#" title="title" data-foo="dataFoo" style="margin: 0;">
        class="yes" goo</a>
    <script>
        var atts = document.querySelector('a');
        console.log(atts.hasAttribute('href'),
                    atts.hasAttribute('title'),
                    atts.hasAttribute('style'),
                    atts.hasAttribute('data-foo'),
                    atts.hasAttribute('class'),
                    atts.hasAttribute('goo'))
    ); // logs "true true true true true"
</script>
</body>
</html>
```

On peut utiliser aussi cette méthode pour vérifier la présence d'un attribut sur un champ de formulaire :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <input type="checkbox" checked></input>
    <script>
        var atts = document.querySelector('input');
        console.log(atts.hasAttribute('checked')) //logs true
    </script>
</body>
</html>
```

4.3.1.6 Récupérer la liste des classes d'un attribut

L'utilisation de la propriété "classList" permet de récupérer une liste, plus précisément une "DOMTokenList".

Il s'agit d'une liste en lecture seule qui contient tous les attributs d'un élément sélectionné, plus quelques informations complémentaires.

Cette liste peut dans certains cas être plus facile à manipuler qu'une liste renvoyée par la propriété className (ou la méthode getAttribute("class")).

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <div class="class1 class2 class3"></div>
    <script>
        var elm = document.querySelector('div');
        console.log(elm.classList); /* logs DOMTokenList {0: "class1",
                                    1: "class2", 2: "class3", length: 3, item: function,
                                    contains: function, add: function, remove: function...} */
        console.log(elm.className); // logs 'class1 class2 class3'
    </script>
</body>
</html>
```

Notes

La propriété classList fonctionne comme un tableau, mais elle est en lecture seule. On peut néanmoins lui appliquer les méthodes add(), move(), contains() et toggle().

IE 9 ne supporte pas la propriété classList, mais IE la supporte.

Exemple d'utilisation des méthodes add() et remove() sur une "classList" :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <div class="dog"></div>
    <script>
        var elm = document.querySelector('div');
        elm.classList.add('cat');
        elm.classList.remove('dog');
        console.log(elm.className); //'cat'
    </script>
</body>
</html>
```

Exemple d'utilisation de la méthode `toggle()` appliquée à une "classList" :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <div class="visible"></div>
    <script>
        var elm = document.querySelector('div');
        elm.classList.toggle('grow');
        console.log(elm.className); // 'visible grow'
        elm.classList.toggle('visible');
        console.log(elm.className); // 'grow'
    </script>
</body>
</html>
```

Exemple d'utilisation de la méthode `contains()` appliquée à une "classList" :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <div class="big brown bear"></div>
    <script>
        var elm = document.querySelector('div');
        console.log(elm.classList.contains('brown')); //logs true
    </script>
</body>
</html>
```

A noter : la propriété `classList` est disponible à tous les niveaux du DOM, on peut par exemple écrire ceci

```
var btn = document.getElementById("action-btn");
btn.addEventListener("click", doSomething, false);

var doSomething = function(event) {
    event.preventDefault();
    if (event.target.parentElement.parentElement.classList.contains("lightbox")) {
        ...
    }
}
```

4.3.1.7 Manipuler les attributs data-*

La propriété "dataset" d'un noeud fournit un objet contenant tous les attributs dont le nom commence par "data-" (on rappelle que ces attributs personnalisés font partie des nouveautés de la norme HTML5).

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <div data-foo-foo="foo" data-bar-bar="bar"></div>
    <script>
        var elm = document.querySelector('div');

        //get
        console.log(elm.dataset.fooFoo); //logs 'foo'
        console.log(elm.dataset.barBar); //logs 'bar'

        //set
        elm.dataset.gooGoo = 'goo';
        console.log(elm.dataset); /* logs DOMStringMap {fooFoo="foo",
                                    barBar="bar", gooGoo="goo"} */

        //what the element looks like in the DOM
        console.log(elm); /* logs <div data-foo-foo="foo" data-bar-bar="bar"
                           data-goo-goo="goo"> */
    </script>
</body>
</html>
```

Notes

Les attributs data-* renvoyés par la propriété "dataset" utilisent la syntaxe dite "camelCase". Cela signifie que "data-foo-foo" sera identifié comme la propriété "fooFoo" dans l'objet "StringMap" associé à la propriété "dataset".

La propriété "dataset" n'est pas supportée par IE 9. On recommandera d'utiliser de préférence les méthodes `getAttribute()`, `removeAttribute()`, `setAttribute()` et `hasAttribute()` décrites dans les chapitres précédents.

4.3.2 Sélectionner des noeuds

4.3.2.1 Sélectionner un élément spécifique

Les méthodes les plus courantes pour sélectionner un noeud "élément" isolé sont les suivantes :

- querySelector()
- getElementById()

Exemples :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <ul>
        <li>Hello</li>
        <li>big</li>
        <li>bad</li>
        <li id="last">world</li>
    </ul>
    <script>
        console.log(document.querySelector('li').textContent); //logs Hello
        console.log(document.querySelector('#last').textContent); //logs world
        console.log(document.getElementById('last').textContent); //logs world
    </script>
</body>
</html>
```

La méthode getElementById() est plus simple d'utilisation que la méthode querySelector(), mais aussi plus limitée, puisqu'elle est spécialisée sur la sélection par "id", comme son nom l'indique.

En plus de permettre des sélections par "id" (il faut dans ce cas faire précéder l'id recherché d'un # dans la chaîne de sélection, comme #last dans l'exemple ci-dessus), la méthode querySelector() permet des sélections plus avancées, en s'appuyant sur les mêmes sélecteurs que CSS3, comme dans l'exemple suivant :

```
console.log(document.querySelector('#liste>ul>li:nth-of-type(2)').textContent);
//logs big
console.log(document.querySelector('#liste>ul>li:nth-of-type(3)').textContent);
//logs bad
```

NB : les sélecteurs utilisés avec querySelector() sont strictement les mêmes que les sélecteurs CSS3. Pour une description de ces sélecteurs, on se reportera au chapitre présentant le principe des sélecteurs CSS, ainsi que le tableau "Sélecteurs CSS3" fourni en annexe.

4.3.2.2 Sélectionner/Créer une NodeList

Les méthodes les plus couramment utilisées pour sélectionner une liste de noeuds dans le DOM sont les suivantes :

- querySelectorAll()
- getElementsByTagName()
- getElementsByClassName()
- getElementsByName()

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <ul id="Liste">
        <li class="liClass">Hello</li>
        <li class="liClass">big</li>
        <li class="liClass">bad</li>
        <li class="liClass">world</li>
    </ul>
    <script>
        /* all of the methods below create/select the same list of <li> elements
           from the DOM */
        console.log(document.querySelectorAll('li'));
        console.log(document.getElementsByTagName('li'));
        console.log(document.getElementsByClassName('liClass'));
    </script>
</body>
</html>
```

NodeList [li.liClass, li.liClass, li.liClass, li.liClass]

HTMLCollection [li.liClass, li.liClass, li.liClass, li.liClass]

HTMLCollection [li.liClass, li.liClass, li.liClass, li.liClass]

Point très important à noter :

La méthode querySelectorAll() renvoie une "NodeList", de même que getElementsByTagName() et getElementsByClassName(), mais dans le cas de ces 2 dernières méthodes, la "NodeList" est une liste "live" qui reflètera toujours l'état exact du document sur lequel la sélection a été effectuée. Cela signifie que si un noeud faisant partie de la sélection est modifié ou supprimé, cette modification sera visible immédiatement dans la "NodeList" renvoyée par les méthodes getElementsByTagName() et getElementsByClassName().

A l'inverse la méthode querySelectorAll() renvoie une "NodeList" statique, qui est un "cliché" de l'état du DOM à un instant donné. Toute modification ou suppression effectuée sur un noeud

faisant partie de la sélection n'affecte en aucune manière la "NodeList" renvoyée par `querySelectorAll()`.

La méthode `getElementsByName()` est intéressante pour obtenir une "NodeList" de tous les éléments ayant le même nom, ce qui peut être intéressant dans le cas de formulaires.

On peut passer le paramètre "*" aux méthodes `querySelectorAll()` et `getElementsByTagName()` pour obtenir une "NodeList" de tous les éléments enfants de l'élément sélectionné. On note également qu'il est possible de chaîner certaines méthodes entre elles, comme dans l'exemple suivant :

```
console.log(document.getElementById('liste').querySelectorAll('*'));
console.log(document.querySelector('#liste').querySelectorAll('*'));
```

`NodeList[li.liClass, li.liClass, li.liClass, li.liClass]`
`NodeList[li.liClass, li.liClass, li.liClass, li.liClass]`

On notera également qu'il est possible d'obtenir la même "NodeList" simplement en sélectionnant l'élément "ul" et en recourant à la propriété `childNodes`, comme dans l'exemple suivant :

```
console.log(document.getElementById('liste').childNodes);
```

Mais le résultat obtenu n'est pas strictement identique, comme en témoigne le contenu de la log :

`NodeList[<TextNode textContent="\n ">,`
`li.liClass, <TextNode textContent="\n ">,`
`li.liClass, <TextNode textContent="\n ">,`
`li.liClass, <TextNode textContent="\n ">,`
`li.liClass, <TextNode textContent="\n ">]`

En revanche, la propriété "children" permet d'obtenir le même résultat :

```
console.log(document.querySelector('#liste').children);
```

`HTMLCollection[li.liClass, li.liClass, li.liClass, li.liClass]`

A noter : Les "NodeLists" se présentent sous la forme de tableaux, dont la propriété "length" (longueur) est en lecture seule. Mais ces tableaux sont particuliers car ils n'héritent pas des méthodes et propriétés de l'objet Array (tableau).

Dans l'exemple précédent, on s'est contenté d'afficher dans la log les "NodeList" récupérées par différentes méthodes.

Si l'on souhaite travailler sur des éléments précis de ces "NodeLists", on peut soit les "adresser" en utilisant la syntaxe spécifique aux tableaux, soit en utilisant la méthode item(), les 2 techniques étant équivalentes.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <ul id="Liste">
        <li class="liClass">Hello</li>
        <li class="liClass">big</li>
        <li class="liClass">bad</li>
        <li class="liClass">world</li>
    </ul>
    <script>
        /* all of the methods below create/select the same list of
           <li> elements from the DOM */
        console.log(document.querySelectorAll('li').item(0).innerHTML );
        // Hello
        console.log(document.querySelectorAll('li')[0].innerHTML ) ; // Hello
        console.log(document.getElementsByTagName('li').item(1).innerHTML ) ;
        // big
        console.log(document.getElementsByTagName('li')[1].innerHTML ) ; // big
        console.log(document.getElementsByClassName('liClass').item(2).innerHTML )
    ); // bad
        console.log(document.getElementsByClassName('liClass')[2].innerHTML ) ;
        // bad
    </script>
</body>
</html>
```

4.3.2.3 Sélectionner des noeuds contextuellement

Les méthodes querySelector(), querySelectorAll(), getElementsByTagName(), et getElementByClassName peuvent être utilisées non seulement sur l'objet "document", mais on peut aussi les utiliser sur la plupart des noeuds du DOM.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
```

```

<div>
  <ul>
    <li class="LiClass">Hello</li>
    <li class="LiClass">big</li>
    <li class="LiClass">bad</li>
    <li class="LiClass">world</li>
  </ul>
</div>
<ul>
  <li class="LiClass">Hello</li>
</ul>
<script>
  /* select a div as the context to run the selecting methods only on the
   contents of the div */
  var div = document.querySelector('div');
  console.log(div.querySelector('ul'));
  console.log(div.querySelectorAll('li'));
  console.log(div.getElementsByTagName('li'));
  console.log(div.getElementsByClassName('liClass'));
</script>
</body>
</html>

```

Ces méthodes opèrent aussi sur des éléments de DOM créés par programmation, comme dans l'exemple suivant :

```

<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
  <script>
    //create DOM structure
    var divElm = document.createElement('div');
    var ulElm = document.createElement('ul');
    var liElm = document.createElement('li');
    liElm.setAttribute('class', 'liClass');
    ulElm.appendChild(liElm);
    divElm.appendChild(ulElm);
    //use selecting methods on DOM structure

    console.log(divElm.querySelector('ul'));
    console.log(divElm.querySelectorAll('li'));
    console.log(divElm.getElementsByTagName('li'));
    console.log(divElm.getElementsByClassName('liClass'));
  </script>
</body>
</html>

```

4.3.2.4 Liste de noeuds préconfigurées

Il est intéressant de noter que plusieurs listes de noeuds préconfigurées sont à la disposition du développeur. Ces listes sont les suivantes :

document.forms	Tous les éléments <form> contenus dans le document HTML
document.images	Tous les éléments contenus dans le document HTML
document.links	Tous les éléments <a> contenus dans le document HTML
document.scripts	Tous les éléments <script> contenus dans le document HTML
document.styleSheets	Tous les éléments <link> ou <style> contenus dans le document HTML

Notes :

Ces tableaux préconfigurés sont construits sur l'objet `HTMLCollection`, excepté pour `document.stylesheets` qui est construit sur l'objet `StyleSheetList`.

Les listes construites à partir de l'objet `HTMLCollection` sont "live", sur le même principe que les "NodeList".

Le raccourci `document.all` est construit à partir de l'objet `HTMLAllCollection` (et non `HTMLCollection`) et n'est pas supporté par Firefox.

4.3.2.5 La méthode matchesSelector()

La méthode matchesSelector() permet de vérifier si un élément correspond bien à un sélecteur.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
  <ul>
    <li>Hello</li>
    <li>world</li>
  </ul>
  <script>
    //fails in modern browser must use browser prefix moz, webkit, o, and ms
    if (document.querySelector('li').matchesSelector) {
      console.log(document.querySelector('li').matchesSelector('li:first-child'));
    }
    //préfixe moz pour Firefox
    if (document.querySelector('li').mozMatchesSelector) {
      console.log(document.querySelector('li').mozMatchesSelector('li:first-child'));
    }
    //préfixe webkit pour Chrome et Safari
    if (document.querySelector('li').webkitMatchesSelector) {
      console.log(document.querySelector('li').webkitMatchesSelector('li:first-child'));
    }
    //préfixe o pour Opera
    if (document.querySelector('li').oMatchesSelector) {
      console.log(document.querySelector('li').oMatchesSelector('li:first-child'));
    }
    //préfixe ms pour IE
    if (document.querySelector('li').msMatchesSelector) {
      console.log(document.querySelector('li').msMatchesSelector('li:first-child'));
    }
  </script>
</body>
</html>
```

A noter :

La méthode matchesSelector() ne bénéficie pas pour l'instant d'un support officiel de la part des navigateurs. Cependant, la plupart l'implémentent à titre expérimental, à condition de la préfixer avec le préfixe spécifique à chaque navigateur. On peut s'assurer que la méthode est implémentée sous l'un des préfixes (moz, webkit, o ou ms) avant de l'invoquer sous ce même préfixe.

La spécification ECMAScript 6 prévoit d'implémenter la méthode matchesSelector() en la renommant en matches().

4.3.2.5 Itération sur une NodeList

On constitue généralement une NodeList afin de pouvoir traiter individuellement chacun des éléments de cette NodeList, par exemple pour appliquer à chacun un style particulier, ou un évènement particulier.

La technique la plus utilisée pour itérer sur une NodeList consiste à utiliser une boucle `for` comme dans l'exemple suivant, dans lequel on considère que la variable "data.items" est une NodeList préchargée avec l'une des méthodes étudiée aux chapitres précédents :

```
var i, i_len, item=null ;
for(i=0, i_len=data.items.length ; i < i_len ; i += 1) {
    item = data.items[i] ;
    /* utiliser la variable item ici */
    if (i > 3) break; /* technique pour forcer l'interruption de la boucle for */
}
```

Les NodeLists ne bénéficient pas nativement d'un mécanisme d'itération équivalent à la fonction `forEach()` que l'on utilise avec les tableaux Javascript. Mais on peut contourner cette limitation en utilisant la technique suivante :

```
[].forEach.call(data.items, function(item) {
    /* utiliser la variable item ici */
});
```

La technique ci-dessus est équivalente à celle proposée par jQuery (ci-dessous), si ce n'est qu'elle est indépendante de tout framework :

```
$.each(data.items, function(i, item) {
    /* utiliser la variable item ici */
})
```

Il n'y a pas de possibilité, avec `foreach()`, de prévoir une condition de sortie équivalente au "break" utilisé dans l'exemple de la boucle "for". Mais on peut contourner la difficulté en filtrant au préalable les éléments de "data.items" sur lesquels on souhaite itérer. Par exemple, si l'on est certain que "data.items" contient au minimum 3 éléments (et que l'on souhaite itérer uniquement sur ces 3 éléments), on peut écrire ceci :

```
[].forEach.call([data.items[0], data.items[1], data.items[2]], function(item) {
    /* utiliser la variable item ici */
});
```

4.3.3 Element Node Style

4.3.3.1 Attribut "style"

Tout élément HTML dispose d'un attribut "style" qui peut être utilisé pour insérer ou modifier des propriétés CSS à la volée.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<body>
    <div style="background-color: red; border: 1px solid black; height: 100px; width: 100px;"></div>
    <script>
        var divStyle = document.querySelector('div').style;
        //logs CSSStyleDeclaration {0="background-color", ...}
        console.log(divStyle);
    </script>
</body>
</html>
```

4.3.3.2 Modifier les attributs CSS en bloc

En utilisant la propriété cssText de l'objet CSSStyleDeclaration, ou en utilisant les méthodes getAttribute(), setAttribute() et removeAttribute(), il est possible de modifier globalement les propriétés de style de tout élément du DOM.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<style>
div {
    background-color: green;
    border: 1px solid black;
    width: 200px;
    height: 200px;
}
</style>
<body>
    <div></div>
    <script>
```

```
var div = document.querySelector('div');
var divStyle = div.style;
//set using cssText
divStyle.cssText = 'background-color:red;border:1px solid
black;height:100px;width:100px;';
//get using cssText
console.log(divStyle.cssText);
//remove
divStyle.cssText = '';
//exactly that same outcome using setAttribute() and getAttribute()
//set using setAttribute
div.setAttribute('style','background-color:red;border:1px solid
black;height:100px;width:100px;');
//get using getAttribute
console.log(div.getAttribute('style'));
//remove
div.removeAttribute('style');

</script>
</body>
</html>
```

4.3.3.3 Modifier les attributs CSS individuellement

Il est possible de modifier à la volée le style d'un élément du DOM.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<style>
div {
    background-color: green ;
    border: 1px solid black ;
    width: 200px ;
    height: 200px ;
}
</style>
<body>

<div id="zone"></div>
<script>
    var divStyle = document.querySelector('div').style;

    // récupération des valeurs de style
    /* toutes les valeurs ci-dessous sont à blanc, les valeurs définies
       au niveau CSS ne sont pas récupérées */
    console.log(divStyle.backgroundColor);
```

```

        console.log(divStyle.border);
        console.log(divStyle.width);
        console.log(divStyle.height);

        /* définition "en ligne", par programmation, du style de la <div>
           Sélectionnée */
        divStyle.backgroundColor = 'red';
        divStyle.border = '1px solid black';
        divStyle.width = '100px';
        divStyle.height = '100px';

        /* récupération des valeurs de style appliquées par programmation
           à la <div> */
        console.log(divStyle.backgroundColor);
        console.log(divStyle.border);
        console.log(divStyle.width);
        console.log(divStyle.height);

        /* suppression des valeurs de style appliqués à la <div> par
           Programmation. Cela a pour effet de restaurer les valeurs
           définies au niveau CSS */
        divStyle.backgroundColor = '';
        divStyle.border = '';
        divStyle.width = '';
        divStyle.height = '';

    </script>
</body>
</html>
```

L'objet "style" est une instance de `CSSStyleDeclaration`. Il fournit un accès aux propriétés CSS, ainsi que plusieurs méthodes très utiles telles que :

- `setProperty(propertyName, value)`
- `getPropertyValue(propertyName)`
- `removeProperty()`

Le code ci-dessous est équivalent au code précédent en utilisant les méthodes `setProperty()`, `getPropertyValue()` et `removeProperty()` :

```

<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<style>
div {
    background-color: green;
    border: 1px solid black;
    width: 200px;
    height: 200px;
}
</style>
<body>
```

```

<div></div>
<script>
    var divStyle = document.querySelector('div').style;

    // récupération des valeurs de style
    /* toutes les valeurs ci-dessous sont à blanc, les valeurs définies au
       niveau CSS ne sont pas récupérées */
    console.log(divStyle.getPropertyValue('background-color'));
    console.log(divStyle.getPropertyValue('border'));
    console.log(divStyle.getPropertyValue('width'));
    console.log(divStyle.getPropertyValue('height'));

    /* définition "en ligne", par programmation, du style de la <div>
       Sélectionnée */
    divStyle.setProperty('background-color', 'red');
    divStyle.setProperty('border', '2px solid green');
    divStyle.setProperty('width', '100px');
    divStyle.setProperty('height', '100px');

    /* récupération des valeurs de style appliquées par programmation
       à la <div> */
    console.log(divStyle.getPropertyValue('background-color'));
    console.log(divStyle.getPropertyValue('border'));
    console.log(divStyle.getPropertyValue('width'));
    console.log(divStyle.getPropertyValue('height'));

    /* suppression des valeurs de style appliqués à la <div> par
       programmation
       cela a pour effet de restaurer les valeurs définies au niveau CSS */
    divStyle.removeProperty('background-color');
    divStyle.removeProperty('border');
    divStyle.removeProperty('width');
    divStyle.removeProperty('height');
</script>
</body>
</html>

```

Points à noter :

Il est important de noter que les styles définis à l'intérieur de la balise `<style>` ne sont pas accessibles par la méthode présentée ci-dessus.

En revanche, les valeurs des éléments de style peuvent être forcées par programmation avec de nouvelles valeurs.

Si on remet à blanc les valeurs de style définis par programmation, alors les valeurs de styles définies à l'intérieur de la balise `<style>` réapparaissent.

Les noms de propriétés définis par programmation obéissent à la syntaxe de type camelCase. Par exemple, "fonsize" devient "fontSize" et "background-image" devient "backgroundImage). Dans le cas particulier où une propriété CSS porte le même nom qu'une propriété Javascript, alors il faut la préfixer avec "css" (par exemple : "float" devient "cssFloat").

Tableau d'équivalence entre propriétés CSS et propriétés Javascript :

background	background
background-attachment	backgroundAttachment
background-color	backgroundColor
background-image	backgroundImage
background-position	backgroundPosition
background-repeat	backgroundRepeat
border	border
border-bottom	borderBottom
border-bottom-color	borderBottomColor
border-bottom-style	borderBottomStyle
border-bottom-width	borderBottomWidth
border-color	borderColor
border-left	borderLeft
border-left-color	borderLeftColor
border-left-style	borderLeftStyle
border-left-width	borderLeftWidth
border-right	borderRight
border-right-color	borderRightColor
border-right-style	borderRightStyle
border-right-width	borderRightWidth
border-style	borderStyle
border-top	borderTop
border-top-color	borderTopColor
border-top-style	borderTopStyle
border-top-width	borderTopWidth
border-width	borderWidth
clear	clear
clip	clip
color	color
cursor	cursor
display	display
filter	filter
font	font
font-family	fontFamily
font-size	fontSize
font-variant	fontVariant
font-weight	fontWeight
height	height
left	left
letter-spacing	letterSpacing
line-height	lineHeight

list-style	listStyle
list-style-image	listStyleImage
list-style-position	listStylePosition
list-style-type	listStyleType
margin	margin
margin-bottom	marginBottom
margin-left	marginLeft
margin-right	marginRight
margin-top	marginTop
overflow	overflow
padding	padding
padding-bottom	paddingBottom
padding-left	paddingLeft
padding-right	paddingRight
padding-top	paddingTop
page-break-after	pageBreakAfter
page-break-before	pageBreakBefore
position	position
float	styleFloat
text-align	textAlign
text-decoration	textDecoration
text-decoration: blink	textDecorationBlink
text-decoration: line-through	textDecorationLineThrough
text-decoration: none	textDecorationNone
text-decoration: overline	textDecorationOverline
text-decoration: underline	textDecorationUnderline
text-indent	textIndent
text-transform	textTransform
top	top
vertical-align	verticalAlign
visibility	visibility
width	width
z-index	zIndex

4.3.3.4 *getComputedStyle()*

La propriété "style" contient le CSS qui est défini par l'attribut de même nom.

Pour accéder au CSS à partir de la cascade de CSS hérités par un élément, il faut utiliser la méthode `getComputedStyle()`. L'information récupérée est en lecture seule, mais elle peut être utilisée pour modifier si besoin les attributs CSS via les méthodes présentées aux chapitres

précédents.

Exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<style>
div {
    background-color: green;
    border: 1px solid purple;
    width: 100px;
    height: 100px;
}
</style>
<body>
    <div></div>
    <script>
        var div = document.querySelector('div');
        //logs "rgb(0, 128, 0)" ou "green", c'est un élément de style "inline"
        console.log(window.getComputedStyle(div).backgroundColor);
        /* logs "1px solid rgb(128, 0, 128)" ou "1px solid purple" : fonctionne
           avec Chrome mais pas avec IE, ni Firefox */
        console.log(window.getComputedStyle(div).border);
        //logs "100px", fonctionne avec IE, Chrome et Firefox
        console.log(window.getComputedStyle(div).width);
        //logs "100px", fonctionne avec IE, Chrome et Firefox
        console.log(window.getComputedStyle(div).height);
    </script>
</body>
</html>
```

4.3.3.5 Ajout de classe et d'attribut à un élément

On peut ajouter une classe à un élément en utilisant les méthodes `setAttribute()` ou `classList.add()`.

La seconde méthode n'est pas reconnue par IE 9, en revanche elle est bien acceptée par Firefox et Chrome.

De même pour les méthodes inverses `removeAttribute()` et `classList.remove()`, la seconde méthode n'est pas non plus reconnue par IE 9.

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<head>
<style>
.foo {
    background-color: red;
```

```
        padding: 10px;
    }

#bar {
    border: 10px solid #000;
    margin: 10px;
}
</style>
</head>
<body>
    <div></div>
    <script>
        var div = document.querySelector('div');

        //set
        div.setAttribute('id', 'bar'); // fonctionne avec tous les navigateurs
        div.classList.add('foo'); // ne fonctionne pas avec IE 9

        /* remove */
        div.removeAttribute('id'); // fonctionne avec tous les navigateurs
        div.classList.remove('foo'); // ne fonctionne pas avec IE 9
    </script>
</body>
</html>
```

Petite photographie de l'objet « classList » associé à un élément d'une page du site W3Schools :

```
> var test = document.querySelector('.w3-left');
< undefined
> console.log(test.classList);
▼ DOMTokenList[2] ⓘ
  0: "w3-navbar"
  1: "w3-left"
  length: 2
  value: "w3-navbar w3-left"
▼ __proto__: DOMTokenList
  ► add: add()
  ► constructor: DOMTokenList()
  ► contains: contains()
  ► entries: entries()
  ► forEach: forEach()
  ► item: item()
  ► keys: keys()
  length: ...
  ► get length: ()
  ► remove: remove()
  ► supports: supports()
  ► toString: toString()
  ► toggle: toggle()
  value: ...
  ► get value: ()
  ► set value: ()
  ► values: values()
  ► Symbol(Symbol.iterator): values()
  Symbol(Symbol.tostringTag): "DOMTokenList"
  ► __proto__: Object
```

On voit que l'élément du DOM sélectionné a 2 classes CSS (w3-navbar et w3-left). L'objet classList associé à cet élément du DOM nous donne accès à différentes méthodes telles que forEach(), remove(), contains(), etc...

4.3.3.6 Ajout de classe et d'attributs à une série d'éléments

Pour ajouter une classe à tous les éléments appartenant à une "NodeList" :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<style>
.one {
    background-color: yellow;
}

.two {
    color: red;
    font-weight: bold;
}
</style>
<body>
    <ul>
        <li class="a">Item</li>
        <li class="b">Item</li>
        <li class="a b">Item</li>
        <li class="c">Item</li>
        <li class="a c">Item</li>
        <li class="b c">Item</li>
    </ul>
    <script>
        var testa = document.querySelectorAll('.a');
        var testb = document.querySelectorAll('.b');
        // ajout classe "one" à toutes les lignes appartenant à la classe "a"
        for ( var i = 0, len = testa.length; i < len; i++) {
            testa[i].classList.add('one');
        }
        // ajout classe "two" à toutes les lignes appartenant à la classe "b"
        for ( var i = 0, len = testb.length; i < len; i++) {
            testb[i].classList.add('two');
        }
    </script>
</body>
</html>
```

4.3.4 Text Nodes

Le texte contenu dans une page HTML est représenté dans le DOM par des instances de la fonction "constructeur" `Text()`, qui produit des "text nodes". Quand un document HTML est "parsé", le texte qui est disséminé à l'intérieur de la page est donc converti en "text nodes", qui

héritent des objets CharacterData, Node et Object.

```
<p>Bonjour</p>

<script>
//select 'hi' text node
var textHi = document.querySelector('p').firstChild ;
console.log(textHi.constructor); //logs [object XrayWrapper function Text() { [native code] }]
console.log(textHi); // <TextNode textContent="Bonjour">
</script>
```

On peut en savoir un peu plus sur les propriétés disponibles en interrogeant le navigateur :

```
<script>
    var text = document.querySelector('p').firstChild;

    /* Affichage des propriétés propres à l'élément sélectionné (fonctionne avec
       Chrome mais pas avec Firefox)
    console.log(Object.keys(text).sort()); /* Chrome renvoie Array[21],
       Firefox renvoie un tableau vide */

    // Affichage des propriétés de l'élément et des propriétés héritées
    var textPropertiesIncludeInherited = [];
    for ( var p in text) {
        textPropertiesIncludeInherited.push(p);
    }
    console.log(textPropertiesIncludeInherited.sort()); // Array[65]

    // Affichage des propriétés héritées uniquement
    var textPropertiesOnlyInherited = [];
    for ( var p in text) {
        if (!text.hasOwnProperty(p)) {
            textPropertiesOnlyInherited.push(p);
        }
    }
    console.log(textPropertiesOnlyInherited.sort()); // Array[44]
</script>
```

Parmi les méthodes les plus intéressantes pour la manipulation d'éléments de type "text node", on trouve :

- `textContent`
- `splitText()`
- `appendData()`
- `deleteData()`
- `insertData()`
- `replaceData()`
- `subStringData()`

- normalize()
- data
- document.createTextNode() (=> ce n'est pas à proprement parler une propriété héritée, mais elle est très utile et sera présentée dans ce chapitre)

Exemple d'utilisation de la méthode createTextNode() :

```
<div></div>
<script>
    var textNode = document.createTextNode('Bonjour');
    document.querySelector('div').appendChild(textNode);
    console.log(document.querySelector('div').innerText); // Firefox renvoie
    "null" alors que Chrome et IE renvoient "Bonjour"
    console.log(document.querySelector('div').innerHTML); // Firefox, Chrome et IE
    renvoient "Bonjour"
</script>
```

Dans l'exemple ci-dessous, l'élément <p> contient 3 noeuds que l'on peut manipuler de différentes manières :

```
<p>Bonjour <strong>Grégory</strong></p>
<script>
    var p_element = document.querySelector('p');
    console.log(p_element.innerHTML); //logs 'Bonjour <strong>Grégory</strong>'
    console.log(p_element.innerText); //logs 'Bonjour Grégory' (sauf sur Firefox
    qui renvoie null)
    console.log(p_element.childNodes.length); //logs 3
    console.log(p_element.firstChild.data); //logs 'Bonjour'
    console.log(p_element.firstChild.nodeValue); //logs 'Bonjour'
    console.log(p_element.firstChild.nextSibling); //logs Element node ou <strong>
    console.log(p_element.firstChild.nextSibling.innerHTML); //logs 'Grégory'
    console.log(p_element.lastChild.innerHTML); //logs 'Grégory'
</script>
```

La méthode textContent() permet de récupérer le contenu d'un élément, et éventuellement de forcer un nouveau contenu sur un élément préalablement sélectionné :

```
<p>Bonjour <strong>Grégory</strong></p>
<script>
    var p_element = document.querySelector('p');
    console.log(p_element.innerHTML); //logs 'Bonjour <strong>Grégory</strong>'
    console.log(p_element.textContent); //logs 'Bonjour Grégory'
    p_element.textContent = p_element.textContent + ', comment vas-tu ?';
    console.log(p_element.innerHTML); //logs 'Bonjour Grégory, comment vas-tu ?'
    (on notera la disparition de <strong>)
</script>
```

REMARQUE : La méthode innerText() ne fait pas partie de spécification HTML5, c'est la raison

pour laquelle certains navigateurs (en particulier Firefox) ne la supportent pas. On recommandera de privilégier l'utilisation de la méthode `textContent()`.

La méthode `normalize()` permet de regrouper en un seul élément de type "Text Node" plusieurs éléments "Text Node" créés dans le DOM par programmation.

```
<div></div>
<script>
    var pElementNode = document.createElement('p');
    var textNode1 = document.createTextNode('Bonjour');
    var textNode2 = document.createTextNode(' ');
    var textNode3 = document.createTextNode('Grégory');
    pElementNode.appendChild(textNode1);
    pElementNode.appendChild(textNode2);
    pElementNode.appendChild(textNode3);
    document.querySelector('div').appendChild(pElementNode);
    console.log(document.querySelector('p').childNodes.length); //logs 3
    document.querySelector('div').normalize(); //combine les 3 noeuds pour n'en
faire qu'un seul
    console.log(document.querySelector('p').childNodes.length); //logs 1
</script>
</body>
```

La méthode `split()` permet de scinder un élément "Text Node" en deux éléments en précisant la position où doit intervenir la séparation :

```
<p>Bonjour Grégory</p>
<script>
    //returns a new text node, taken from the DOM
    console.log(document.querySelector('p').firstChild.splitText(8).data); //logs
Grégory
    //What remains in the DOM...
    console.log(document.querySelector('p').firstChild.textContent); //logs
Bonjour
</script>
```

4.4 Javascript dans le DOM

4.4.1 Insertion et exécution du code Javascript

Le code Javascript peut être inséré à l'intérieur d'une page web de différentes manières :

1 - le code peut être inséré directement dans la page en étant placé entre les balises `<script>` et `</script>`. Exemple :

```
<script> alert('coucou me revoilou !') ; </script>
```

2 - le code peut être placé dans un fichier externe ayant pour extension `.js`, et inséré dans la page au moyen du code suivant :

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

3 - le code Javascript peut être disséminé dans le code HTML via des évènements attachés à des éléments HTML, comme par exemple le clic de souris :

```
<button onclick="doSomething()" id="action-btn">Cliquez moi !</button>
```

La première solution est pratique dans un contexte pédagogique, pour s'initier au Javascript, comme on le fait dans le cadre de ce cours.

La seconde solution est nettement préférable à la première pour le développement d'applications professionnelles.

La troisième solution, qui a été longtemps pratiquée, n'est plus considérée à l'heure actuelle comme une bonne pratique.

Voici à titre d'information de quelle manière on attache un évènement à un élément HTML sur différents frameworks Javascript :

```
// jQuery
$("#action-btn").on("click", doSomething);
// YUI
Y.one("#action-btn").on("click", doSomething);
// Dojo
var btn = dojo.byId("action-btn");
dojo.connect(btn, "click", doSomething);
```

4.4.2 Le timing du téléchargement de code

Par défaut, lorsque le navigateur analyse le DOM et qu'il rencontre un élément `<script>`, il stoppe l'analyse du document et toutes les actions en cours (y compris les téléchargements), et exécute le code Javascript. Ce comportement est dit "synchrone". Si le Javascript est externe au document HTML, ce comportement est encore plus pénalisant car le code doit être téléchargé, puis exécuté, avant que le navigateur puisse reprendre le traitement des opérations restantes.

Si le navigateur doit télécharger de gros fichiers Javascript dont les déclarations sont situées en début de page, l'incidence sur le chargement global de la page peut être suffisamment significative pour que l'utilisateur ait l'impression de perdre son temps.

Le mot clé "defer" permet de forcer le navigateur à différer le chargement d'un fichier Javascript :

```
<script defer type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/10.1/jquery.min.js"></script>
```

D'après la spécification du W3C, les scripts dont le téléchargement est différé sont sensés s'exécuter juste avant l'exécution de l'évènement `DOMContentLoaded`. Dans la pratique, tous les navigateurs n'ont pas un comportement homogène par rapport au mot clé "defer", même si la situation tend à se normaliser sur les versions les plus récentes des navigateurs.

L'élément `<script>` a un attribut appelé "async" qui indique au navigateur de ne pas bloquer la construction de la page HTML pendant le téléchargement des fichiers Javascript. Lorsque l'on utilise l'attribut "async", les fichiers sont chargés en parallèle et leur analyse (la phase boquante) ne démarre que quand ils sont entièrement téléchargés.

```
<script async type="text/javascript"
src="http://ajax.googleapis.com/ajax/libs/jquery/10.1/jquery.min.js"></script>
```

L'attribut "async" l'emporte sur l'attribut "defer" si les deux sont déclarés à l'intérieur d'un élément "script".

Le support de l'attribut "async" n'est pas assuré par IE9, mais il le sera avec IE10.

L'attribut "async" peut poser des problèmes si certains fichiers Javascript sont dépendants d'autres fichiers (qui doivent impérativement être chargés avant) pour leur bon fonctionnement.

Une astuce souvent utilisée pour forcer un navigateur à télécharger de manière asynchrone du code JavaScript sans utiliser l'attribut `async` est d'effectuer le téléchargement par programmation. La technique consiste à créer par programmation un nœud `<script>`, en précisant l'url du fichier à télécharger via l'attribut "src", puis à ajouter cet élément `<script>` à

l'élément <body>. Cette technique force le navigateur à traiter l'élément <script> de manière asynchrone :

```
<script>
  var jqueryScript = document.createElement("script");
  jqueryScript.src =
    "http://ajax.googleapis.com/ajax/libs/jquery/10.1/jquery.min.js";
  document.body.appendChild(jqueryScript);
</script>
```

ATTENTION : pour que le téléchargement asynchrone n'entraîne pas de perturbation dans le code Javascript, il faut à minima que l'exécution du code soit placée sur l'évènement " onload" associé à l'élément "body" de la page :

```
<script>
  document.body.onload = function() {
    var test = jQuery('li lvl2:parent');
    console.log(test); // Object[li lvl2, li lvl2, li lvl2]
  }
</script>
```

Une autre solution consiste à ajouter un attribut " onload" sur le noeud <script> qui a été créé par programmation, et à intégrer sur cet attribut " onload" une fonction anonyme contenant le code qui doit être exécuté impérativement après le téléchargement du code Javascript externe :

```
<script>
  var jqueryScript = document.createElement("script");
  jqueryScript.src =
    "http://ajax.googleapis.com/ajax/libs/jquery/10.1/jquery.min.js";
  jqueryScript.onload = function() {
    var test2 = jQuery('li lvl2:parent');
    console.log(test2); // Object[li lvl2, li lvl2, li lvl2]
  }
  document.body.appendChild(jqueryScript);
</script>
```

Malheureusement, la technique ci-dessus est faiblement supportée par les navigateurs. Seul Firefox l'accepte à l'heure actuelle. On n'en recommandera donc pas l'usage.

Nous verrons dans l'un des chapitres suivants (qui s'intitule "DOM Events") une meilleure manière de déclencher le chargement du code Javascript, avec l'utilisation du DOMContentLoadedEvent.

4.4.3 Recommandations

Compte tenu de la nature synchrone du chargement et de l'exécution du code Javascript, on recommandera en règle générale de placer les éléments <script>, qu'ils soient liés à du code interne ou externe, en fin de page HTML, juste avant la balise </body>. En procédant de la sorte, on est sûr que l'essentiel de la page sera chargé avant que le navigateur ne procède au téléchargement et au parsing du code Javascript.

Il est à ce titre non recommandé d'utiliser la technique d'inclusion de code Javascript suivante (présentée au début du chapitre) :

```
<button onclick="doSomething()" id="action-btn">Click Me</button>
```

En effet, si un utilisateur clique sur le bouton "Click Me" avant que le chargement du code Javascript contenant la fonction doSomething() ne soit achevé, cette action de l'utilisateur déclenchera un plantage et donc un blocage de la page. On verra au chapitre suivant qu'il est largement préférable d'utiliser la gestion des événements pour associer une action à un clic de souris par exemple.

4.5 DOM Events

4.5.1 Introduction

On a vu au chapitre précédent que beaucoup de développeurs utilisaient la solution de facilité suivante pour associer un élément HTML à un évènement :

```
<button onclick="doSomething();return false;" id="action-btn">Cliquez moi !</button>
```

Mais nous avons vu qu'il s'agissait d'une mauvaise pratique et nous avons expliqué pourquoi.

Voyons tout de suite un exemple de bonne pratique :

```
<button id="action-btn">Cliquez moi !</button>

<script>
    function doSomething(evt) {
        console.log(evt.clientX); /* 79 (dépend de la position du bouton et
                                    de la position du curseur lors du clic) */
        console.log(evt.clientY); // 205 (idem)
        console.log(evt.currentTarget); // <button id="action-btn">
        console.log(evt.currentTarget.nodeName); // BUTTON
        console.log(evt.currentTarget.getAttribute('id')); // action-btn
        console.log(evt.currentTarget.parentNode); // <body>
        console.log(evt.currentTarget.children); // HTMLCollection[]

        console.log(evt.currentTarget.childNodes);
        // NodeList[<TextNode.textContent="Cliquez moi !"]>
        evt.currentTarget.style.color = '#F00';
        // Le texte du bouton passe en rouge
    }
    var btn = document.getElementById("action-btn");
    btn.addEventListener("click", doSomething, false);
</script>
```

La méthode `addEventListener()` a été normalisée récemment par le W3C mais elle est antérieure à cette norme, et bénéficie d'ores et déjà d'un bon support par la plupart des navigateurs, sauf pour les versions de IE antérieures à la 9. Pour que le code ci-dessus fonctionne avec IE 8 et les versions précédentes, il faut écrire ceci :

```
if (btn.addEventListener) {
    // solution compatible tous navigateurs sauf IE < 9
    btn.addEventListener("click", doSomething, false);
} else if (btn.attachEvent) {
    // pour IE < 9
    btn.attachEvent("onclick", doSomething);
} else {
    // solution de la dernière chance
    btn["onclick"] = handler;
}
```

Pour éviter de réinventer la roue à chaque fois, on aura tout intérêt à créer une fonction générique telle que celle ci-dessous :

```
function addListener(eventObj, eventName, eventHandler) {
    if (eventObj.addEventListener) {
        // solution compatible tous navigateurs sauf IE < 9
        return eventObj.addEventListener(eventName, eventHandler, false);
    } else if (eventObj.attachEvent) {
        // pour IE < 9
        return eventObj.attachEvent("on" + eventName, eventHandler);
    } else {
        // solution de la dernière chance
        return eventObj["on" + eventName] = eventHandler;
    }
}
```

Fonction que l'on pourra appeler via le code suivant :

```
var btn = document.getElementById("action-btn");
addListener(btn, 'click', doSomething, false);
```

A noter qu'il existe une méthode plus simple et compatible tous navigateurs pour attacher un évènement de type "click" sur un bouton :

```
var btn = document.getElementById("action-btn");

// pour attacher un évènement à un élément, 2 syntaxes possibles, strictement
équivalentes :
// - syntaxe 1 :
btn['onclick'] = doSomething ;
// - syntaxe 2 :
btn.onclick = doSomething ;
```

On notera que la syntaxe 1 est la même syntaxe qui est utilisée, comme solution de la "dernière

chance" dans la fonction addListener() présentée ci-dessus.

Pourquoi utiliser addEventListener() plutôt que la technique plus simple ci-dessus ? Parce que addEventListener() permet d'associer plusieurs fonctions à un même élément et un même type d'évènement, comme dans l'exemple suivant :

```
var btn = document.getElementById("action-btn");
addEventListener(btn, 'click', function(){alert(1)}, false) ;
addEventListener(btn, 'click', function(){alert(2)}, false) ;
```

Petit comparatif avec l'affectation d'évènements telle qu'elle est pratiquée dans certains frameworks :

```
// jQuery
$("#action-btn").on("click", doSomething);
// YahooUI
Y.one("#action-btn").on("click", doSomething);
// Dojo
var btn = dojo.byId("action-btn");
dojo.connect(btn, "click", doSomething);
```

Quelques précisions sur les paramètres de la méthode addEventListener() :

Les paramètres sont dans l'ordre:

1. Le nom de l'évènement (ce nom est une DOMString).
2. Le nom d'une fonction appelée lorsque l'évènement est déclenché (on l'appelle un "Event Listener").
3. La valeur true pour le mode capture ou false pour le mode bubbling (voir définitions sur la page suivante).

- Le premier argument est une chaîne qui reprend le nom d'un évènement standard. Les évènements reconnus pour la souris sont :

- click
- mousedown
- mouseup
- mouseover
- mousemove
- mouseout

Pour le clavier (DOM 3):

- keydown
- keypress (attention : toutes les touches ne sont pas reconnues)
- keyup

DOM 3 reconnaît d'autres types d'évènements, le lien ci-dessous propose une liste exhaustive :
<http://help.dottoro.com/larrqck.php>

- La fonction en second paramètre est un EventListener (écouteur d'évènement) : Le "listener" peut être une fonction définie par l'utilisateur ou un objet implémentant l'interface EventListener. Pour passer des paramètres à la fonction, on utilise une fonction anonyme. Il est néanmoins préférable de passer un nom de fonction, ce qui facilitera l'utilisation de la fonction removeEventListener(), qui sera présentée plus loin.
- Le troisième paramètre est une valeur booléenne : elle concerne la capture des évènements du type donné. On mettra "false" si on n'en veut pas (c'est la valeur par défaut). Si la valeur est vraie, alors la propagation de l'évènement se fera en commençant par les balises conteneurs de plus haut niveau, avant de se propager jusqu'à l'élément cible.

Capture et remontée (bubbling)

On parle de capture quand un évènement est traité par la balise conteneur avant d'être transmise à la balise contenue. On parle de remontée (bubbling en anglais) quand l'ordre inverse se produit, ce qui est le standard quand on ne spécifie rien dans le code. Car la propriété addEventListener dispose d'un paramètre pour décider de l'ordre. Si le troisième paramètre vaut true, on est en mode "capture", s'il est false, on est en mode "bubbling".

Suppression d'un écouteur d'évènement :

```
var btn = document.getElementById("action-btn");
btn.addEventListener('click', foo, false);
btn.removeEventListener('click', foo, false);
```

On peut créer une fonction générique de suppression, sur le même principe que la fonction addListener() :

```
function removeListener(eventObj, eventName, eventHandler) {
    if (eventObj.removeEventListener) {
        eventObj.removeEventListener(eventName, eventHandler, false);
    } else if (eventObj.detachEvent) {
        eventObj.detachEvent("on" + eventName, eventHandler);
    } else {
        eventObj["on" + eventName] = null;
    }
}
```

ATTENTION : pour supprimer un écouteur d'évènement, il est impératif de le faire en transmettant à la méthode removeEventListener() les mêmes paramètres qui ont servi à créer

l'écouteur d'évènement. C'est la raison pour laquelle il est préférable de transmettre en second paramètre un nom de fonction, plutôt que le code d'une fonction anonyme.

On peut considérer comme une bonne pratique le fait de regrouper les fonctions addListener() et removeListener() à l'intérieur d'une même classe (ici MyEventManager) :

```
var MyEventManager = {
    addListener: function(eventObj, eventName, eventHandler) {

        if (eventObj.addEventListener) {
            // solution compatible tous navigateurs sauf IE < 9
            return eventObj.addEventListener(eventName, eventHandler,
                false);
        } else if (eventObj.attachEvent) {
            // pour IE < 9
            return eventObj.attachEvent("on" + eventName, eventHandler);
        } else {
            // solution de la dernière chance
            return eventObj["on" + eventName] = eventHandler;
        }
    },
    removeListener: function(eventObj, eventName, eventHandler) {
        if (eventObj.removeEventListener) {
            eventObj.removeEventListener(eventName, eventHandler,
                false);
        } else if (eventObj.detachEvent) {
            eventObj.detachEvent("on" + eventName, eventHandler);
        } else {
            eventObj["on" + eventName] = null;
        }
    }
}
```

4.5.2 Référencer la cible d'un évènement

Il est possible d'affecter un écouteur d'évènement à un élément "global" de manière à affecter cet écouteur à tous les enfants de l'élément global.

Cet élément global peut être un tableau HTML, l'écouteur s'appliquant dans ce cas à toutes les cellules du tableau. Ce peut être une "div" et dans ce cas l'évènement pourrait s'appliquer à chacun des paragraphes de cette "div".

Premier exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<head>
<title>Insert title here</title>
</head>
<body>
    <div>
        Cliquez sur moi
        <p>Paragraphe 1</p>
        <p>Paragraphe 2</p>
    </div>
    <script>
        document.body.addEventListener('click', function(event) {
            console.log(event.target);
            if (event.target.tagName.toLowerCase() === 'p') {
                console.log('bingo !')
            } else {
                console.log('loupé !')
            }
        }, false);
    </script>
</body>
</html>
```

Second exemple :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<head>
<title>Insert title here</title>
<style>
table {
    max-width: 100%;
    background-color: transparent;
    border-collapse: collapse;
    border-spacing: 0;
}
th,
td {
    padding: 8px;
    line-height: 20px;
    text-align: left;
    vertical-align: top;
    border-top: 1px solid #dddddd;
}
.degrade {
    background-color: #f5f5f5;
```

```
border: 1px solid #e3e3e3;
-webkit-border-radius: 4px;
-moz-border-radius: 4px;
border-radius: 4px;
-webkit-box-shadow: inset 0 1px 1px rgba(0, 0, 0, 0.05);
-moz-box-shadow: inset 0 1px 1px rgba(0, 0, 0, 0.05);
box-shadow: inset 0 1px 1px rgba(0, 0, 0, 0.05);
background-color: #0044cc;
background-image: -moz-linear-gradient(top, #0088cc, #0022cc);
background-image: -webkit-gradient(linear, 0 0, 0 100%, from(#0088cc),
to(#0022cc));
background-image: -webkit-linear-gradient(top, #0088cc, #0022cc);
background-image: -o-linear-gradient(top, #0088cc, #0022cc);
background-image: linear-gradient(to bottom, #0088cc, #0022cc);
}

.fade {
    border: 1px solid #f5f5f5;
    -webkit-box-shadow: inset 0 2px 4px rgba(0, 0, 0, 0.15), 0 1px 2px rgba(0, 0, 0,
0.05);
    -moz-box-shadow: inset 0 2px 4px rgba(0, 0, 0, 0.15), 0 1px 2px rgba(0, 0, 0,
0.05);
    box-shadow: inset 0 2px 4px rgba(0, 0, 0, 0.15), 0 1px 2px rgba(0, 0, 0,
0.05);
    -webkit-transition: height 0.95s ease;
    -moz-transition: height 0.95s ease;
    -o-transition: height 0.95s ease;
    transition: height 0.95s ease;
}

td:hover {
    color: #000000;
    text-decoration: none;
    cursor: pointer;
    opacity: 0.4;
    filter: alpha(opacity=40);
}

</style>
</head>
```

```

<body>
    <p>Cliquer dans les cellules du tableau</p>
    <table border="1">
        <tbody>
            <tr>
                <td>ligne 1 colonne 1</td>
                <td>ligne 1 colonne 2</td>
            </tr>
            <tr>
                <td>ligne 2 colonne 1</td>
                <td>ligne 2 colonne 2</td>
            </tr>
            <tr>
                <td>ligne 3 colonne 1</td>
                <td>ligne 3 colonne 2</td>
            </tr>
            <tr>
                <td>ligne 4 colonne 1</td>
                <td>ligne 4 colonne 2</td>
            </tr>
            <tr>
                <td>ligne 5 colonne 1</td>
                <td>ligne 5 colonne 2</td>
            </tr>
            <tr>
                <td>ligne 6 colonne 1</td>
                <td>ligne 6 colonne 2</td>
            </tr>
        </tbody>
    </table>
    <script>

        document.querySelector('table').addEventListener('click',
            function(event) {
                if (event.target.tagName.toLowerCase() === 'td') {
                    console.log(event.target.textContent);
                    event.target.textContent =
                        event.target.textContent.toUpperCase();
                    if (event.target.classList.contains('degrade')) {
                        event.target.classList.remove('degrade');
                        event.target.classList.add('fade');
                    } else {
                        event.target.classList.add('degrade');
                    }
                }
            },
            false);
    </script>
</body>
</html>

```

4.5.3 Stopper le flow standard avec preventDefault()

Les navigateurs gèrent un certain nombre d'évènements attribués par défaut à certains types d'éléments HTML.

Par exemple, le fait de cliquer sur une balise `<a>` a pour effet de déclencher une requête HTTP de type GET vers une URL définie dans l'attribut "href".

Ce comportement par défaut de la balise `<a>` peut être bloqué au moyen de la méthode `preventDefault()`, comme dans l'exemple suivant :

```
<a href="google.com">no go</a>
<input type="submit" id="input_submit" />
<script>
    document.querySelector('a').addEventListener('click', function(event) {
        event.preventDefault();
        console.log('preventDefault() appliquée à la balise <a>, stoppe le
comportement par défaut de la balise qui est de charger une URL');
    }, false);

    document.querySelector('#input_submit').addEventListener('click',
        function(event) {
            event.preventDefault();
            console.log('preventDefault() appliquée à un bouton
"submit", ce qui stoppe son comportement par défaut consistant à envoyer une requête
HTTP' );
    }, false);

    document.body.addEventListener('click', function() {
        console.log('le listener attaché à l\'élément <body> n\'est pas affecté
par les preventDefault() appliqués à ses "enfants"');
    }, false);
</script>
```

4.5.4 Bloquer la propagation d'un évènement

L'appel de la méthode stopPropagation() sur un évènement a pour effet de stopper la capture de l'évènement qui ne remontera pas aux écouteurs d'évènement liés à des parents de l'élément cible.

Exemple :

```
<div>click me</div>
<script>
    document.querySelector('div').addEventListener('click',function(){
        console.log('propagation non interceptée sur cet écouteur
d\'évènement');
    },false);

    document.querySelector('div').addEventListener('click',function(event){
        console.log('propagation interceptée sur cet écouteur d\'évènement');
        event.stopPropagation();
        //event.stopImmediatePropagation();
    },false);

    document.querySelector('div').addEventListener('click',function(){
        console.log('Ce troisième écouteur est appelé si event.stopPropagation()
est utilisée, mais pas si la méthode event.stopImmediatePropagation() est appelée');
    },false);

    document.body.addEventListener('click',function(){
        console.log('cet écouteur ne peut être appelé, la propagation étant
arrêtée par un écouteur sur le même type d\'évènement');
    },false);
</script>
```

Conseil : mettez en commentaire la ligne "event.stopPropagation()" et regardez ce qui se produit. Puis remplacez par la ligne "event.stopImmediatePropagation()", et regardez la différence.

4.5.5 Evénement Touch

Avec l'augmentation exponentielle du nombre de terminaux mobiles utilisant des applications webs, il devenait nécessaire de disposer d'une API adaptée dédiée aux écrans tactiles.

Pour répondre à ce besoin spécifiquement, le W3C a développé une première version de spécification appelée "Touch Events Level 1", qui définit 4 types d'évènement :

- "touchstart" qui se déclenche quand un doigt touche l'écran.
- "touchend" qui se déclenche quand le doigt "quitte" l'écran
- "touchmove" qui se déclenche quand un doigt se déplace sur l'écran entre 2 points
- "touchcancel" qui se déclenche quand le navigateur interrompt l'action de "toucher", ou quand le doigt quitte la zone active de l'écran

Une seconde version de la spécification, appelée "Touch Events Level 2", qui est encore en cours de développement, introduit 2 évènements supplémentaires :

- "touchenter" qui se déclenche lorsqu'un doigt, déjà en contact avec l'écran, se déplace vers un élément cible
- "touchleave" qui se déclenche quand le doigt "sort" de la zone cible sans quitter l'écran

Exemple :

```
window.addEventListener('touchstart', function () {...}, false);
```

Chaque fois qu'un événement "tactile" est déclenché, il crée un objet "TouchEvent", qui contient un certain nombre d'informations sur l'événement, dont une "TouchList" indiquant l'enchaînement des actions de toucher effectuées par l'utilisateur.

Pour une présentation détaillée :

https://developer.mozilla.org/fr/docs/Web/Guide/DOM/Events/Touch_events

4.5.6 Evénement Load, les bonnes pratiques

On souhaite souvent mettre en place des écouteurs d'évènement sur certains éléments d'une page, mais se pose la question de savoir quand mettre en place ces écouteurs d'évènements. En toute logique, la bonne pratique consiste à mettre en place ces écouteurs d'évènement une fois que le DOM est complètement « chargé ». Mais comment savoir si le DOM est « chargé » ?

L'ancienne manière de charger le code Javascript dans une page consistait à utiliser l'évènement "load" associé à l'élément HTML « body ».

```
<html> ...
  <body onload="myOnloadFunc();">
</html>
```

... ou encore ...

```
<script>
  document.body.onload = myOnloadFunc ;
</script>
```

... ou encore ...

```
<script>
  document.body.addEventListener ( 'load', myOnloadFunc, false);
</script>
```

... ou encore ...

```
<script>
  window.onload = myOnloadFunc ;
</script>
```

... ou encore ...

```
<script>
  window.addEventListener ( 'load', myOnloadFunc, false);
</script>
```

On constate donc que certains développeurs déclenchent l'appel de leur fonction de mise en place des écouteurs d'évènements sur l'objet body associé à l'élément HTML de même nom, via l'évènement « load ». D'autres effectuent cette même mise en place sur l'objet window. Si les deux approches semblent équivalentes, il y a cependant une différence :

- `document.body.onload` : l'évènement « load » est déclenché dès que le DOM est chargé (c'est-à-dire avant même le chargement de l'ensemble des fichiers annexes).
- `window.onload` : l'évènement « load » est déclenché quand toute la page est chargée (et

pas seulement le DOM), ce qui inclut les fichiers CSS, JS, images, etc... Il peut en résulter un temps de latence désagréable pour les utilisateurs.

L'évènement « load » associé à « document.body.onload » est donc susceptible de se déclencher plus tôt que le même évènement associé à « window.onload ». Mais dans la pratique cela n'est pas vrai pour tous les navigateurs. En effet, il semble que certains navigateurs déclenchent l'évènement « load » sur « window.onload » même si tous les fichiers ne sont pas encore chargés. Difficile de s'y retrouver dans ce contexte...

Plus récemment est apparu un évènement de type « DOM ready », destiné à clarifier la situation. Cet événement a rencontré un certain succès auprès des développeurs de librairies et de frameworks, à tel point qu'il a été intégré en natif dans les navigateurs au travers de l'évènement DOMContentLoaded, et qu'il fait maintenant partie de la norme HTML5.

L'évènement « DOMContentLoaded » est déclenché sur l'objet "document" de la façon suivante:

```
<script>
    document.addEventListener ( 'DOMContentLoaded', myOnloadFunc, false);
</script>
```

Cette solution permet à l'utilisateur d'obtenir "la main" plus rapidement, et elle est particulièrement intéressante sur des connexions lentes, ou lorsque les pages sont "lourdes". Elle est bien supportée par les navigateurs récents supportant la norme HTML5.

A noter que jQuery fournit une structure équivalente avec la méthode « ready ». Dans les coulisses, la méthode « ready » regarde si l'objet DOMContentLoaded est disponible sur le navigateur, auquel cas elle s'en sert, dans le cas contraire elle met en place un eventListener sur l'évènement « load ».

```
$(document).ready(myOnloadFunc) ;
```

Il est important de noter que la technique du « window.onload » a des effets de bord problématiques. En effet, un seul « window.onload » peut être traité par le navigateur à un instant T. Or, dans certaines applications complexes, les développeurs sont quelquefois tentés de placer un « window.onload » dans plusieurs fichiers, créant un risque potentiel de conflit. Ce problème est bien expliqué dans cet article :

<http://www.webreference.com/programming/javascript/onloads/index.html>

Lee Underwood, l'auteur de cet article, propose une solution de contournement qui est la suivante :

```
function addLoadEvent(func) {  
    var oldonload = window.onload;  
    if (typeof window.onload != 'function') {  
        window.onload = func;  
    } else {  
        window.onload = function() {  
            if (oldonload) {  
                oldonload();  
            }  
            func();  
        }  
    }  
}  
addLoadEvent(nameOfSomeFunctionToRunOnPageLoad);  
addLoadEvent(function() {  
    /* more code to run on page load */  
});
```

En règle générale, on va utiliser l'évènement "load" pour mettre en place des écouteurs d'évènements relatifs à des clics sur des liens (balises "a"), ou à des clics sur des boutons (balises "button" et balises "input" de type "button" ou "submit"). Cela signifie que ces clics ne seront opérationnels qu'une fois le DOM mis en place. Pour éviter que les internautes ne soient tentés de cliquer sur des boutons et des liens qui ne sont pas opérationnels, il est souhaitable de désactiver temporairement la possibilité de cliquer sur ces éléments. Pour ce faire, on peut procéder de la façon suivante :

- mettre en place un attribut « disabled » sur les boutons concernés, et ajouter un attribut « data-btncliquable » ce qui permettra de sélectionner les éléments plus facilement pour les activer
- Pour les balises « a », il faut utiliser une classe CSS permettant de rendre les liens concernés inopérants, avec ajout d'un attribut « data-liencliquable », ce qui permettra de les identifier plus facilement pour la suppression de la classe concernée

Voici un exemple d'implémentation incluant le CSS, le HTML et le JS :

```
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style>
        .inactiveLink {
            pointer-events: none;
            cursor: default;
        }
    </style>
</head>
<body>
    <button id="titi" data-btncliquable="true" disabled>titi</button>
    <button id="titi" data-btncliquable="true" disabled>gros minet</button>

    <a href="https://fr.wikipedia.org/wiki/Wikip%C3%A9dia:Accueil_principal"
data-liencliquable="true" target="_blank" class="inactiveLink" >lien cliquable
1</a>
    <a href="https://fr.wikipedia.org/wiki/Wikip%C3%A9dia:Accueil_principal"
data-liencliquable="true" target="_blank" class="inactiveLink" >lien cliquable
2</a>

<script>
    window.addEventListener('load', function() {
        // activation des boutons
        var btnclicks = document.querySelectorAll("[data-btncliquable]");
        for (var i = 0, imax = btnclicks.length; i < imax; i++) {
            btnclicks[i].removeAttribute('disabled');
        }
        // activation des liens
        var linkclicks = document.querySelectorAll("[data-liencliquable]");
        for (var i = 0, imax = linkclicks.length; i < imax; i++) {
            linkclicks[i].classList.remove('inactiveLink');
        }
    }, false);
</script>
</body>
</html>
```

On retrouve cet exemple en live sur codepen :

<http://codepen.io/gregja/pen/EWJXbo/>

4.5 Les objets Window et Screen

4.5.1 Introduction

Nous avons évoqué à plusieurs reprises l'objet Window, pour différents usages (cf. fonction NaN, ou encore le chapitre relatif aux mediaqueries).

Cet objet est disponible sur tous les navigateurs, et il fournit un ensemble de propriétés et de méthodes qui peuvent se révéler très pratiques. Cet objet n'est en revanche pas disponible en environnement NodeJS (côté serveur).

Je vous recommande d'utiliser la fonction console.log() sur l'objet « window », vous vous rendrez compte que cet objet centralise un nombre impressionnant de propriétés et méthodes.

Les tableaux de propriétés et méthodes ci-dessous proviennent du site W3Schools. Ils ne sont pas du tout exhaustifs, mais ils proposent une sélection intéressante :

http://www.w3schools.com/jsref/obj_window.asp

Property	Description
closed	Returns a Boolean value indicating whether a window has been closed or not
defaultStatus	Sets or returns the default text in the statusbar of a window
document	Returns the Document object for the window (See Document object)
frameElement	Returns the <iframe> element in which the current window is inserted
frames	Returns all <iframe> elements in the current window
history	Returns the History object for the window (See History object)
innerHeight	Returns the inner height of a window's content area
innerWidth	Returns the inner width of a window's content area
length	Returns the number of <iframe> elements in the current window
localStorage	Returns a reference to the local storage object used to store data. Stores data with no expiration date
location	Returns the Location object for the window (See Location object)
name	Sets or returns the name of a window
navigator	Returns the Navigator object for the window (See Navigator object)
opener	Returns a reference to the window that created the window
outerHeight	Returns the outer height of a window, including toolbars/scrollbars
outerWidth	Returns the outer width of a window, including toolbars/scrollbars
pageXOffset	Returns the pixels the current document has been scrolled (horizontally) from the upper left corner of the window
pageYOffset	Returns the pixels the current document has been scrolled (vertically) from the upper left corner of the window

<u>parent</u>	Returns the parent window of the current window
<u>screen</u>	Returns the Screen object for the window (See Screen object)
<u>screenLeft</u>	Returns the horizontal coordinate of the window relative to the screen
<u>screenTop</u>	Returns the vertical coordinate of the window relative to the screen
<u>screenX</u>	Returns the horizontal coordinate of the window relative to the screen
<u>screenY</u>	Returns the vertical coordinate of the window relative to the screen
<u>sessionStorage</u>	Returns a reference to the local storage object used to store data. Stores data for one session (lost when the browser tab is closed)
<u>scrollX</u>	An alias of pageXOffset
<u>scrollY</u>	An alias of pageYOffset
<u>self</u>	Returns the current window
<u>status</u>	Sets or returns the text in the statusbar of a window
<u>top</u>	Returns the topmost browser window
Method	Description
<u>alert()</u>	Displays an alert box with a message and an OK button
<u>atob()</u>	Decodes a base-64 encoded string
<u>blur()</u>	Removes focus from the current window
<u>btoa()</u>	Encodes a string in base-64
<u>clearInterval()</u>	Clears a timer set with setInterval()
<u>clearTimeout()</u>	Clears a timer set with setTimeout()
<u>close()</u>	Closes the current window
<u>confirm()</u>	Displays a dialog box with a message and an OK and a Cancel button
<u>focus()</u>	Sets focus to the current window
<u>getComputedStyle()</u>	Gets the current computed CSS styles applied to an element
<u>getSelection()</u>	Returns a Selection object representing the range of text selected by the user
<u>matchMedia()</u>	Returns a MediaQueryList object representing the specified CSS media query string
<u>moveBy()</u>	Moves a window relative to its current position
<u>moveTo()</u>	Moves a window to the specified position
<u>open()</u>	Opens a new browser window
<u>print()</u>	Prints the content of the current window
<u>prompt()</u>	Displays a dialog box that prompts the visitor for input
<u>resizeBy()</u>	Resizes the window by the specified pixels
<u>resizeTo()</u>	Resizes the window to the specified width and height
<u>scroll()</u>	Deprecated, replaced by ScrollTo()
<u>scrollBy()</u>	Scrolls the document by the specified number of pixels
<u>scrollTo()</u>	Scrolls the document to the specified coordinates
<u>setInterval()</u>	Calls a function or evaluates an expression at specified intervals (in milliseconds)
<u>setTimeout()</u>	Calls a function or evaluates an expression after a specified number of milliseconds
<u>stop()</u>	Stops the window from loading

La propriété “location” de l’objet Window fournit plusieurs informations importantes :

```
> console.log(window.location);
▼ Location ⓘ
  ► ancestorOrigins: DOMStringList
  ► assign: ()
  hash: ""
  host: "www.w3schools.com"
  hostname: "www.w3schools.com"
  href: "http://www.w3schools.com/jsref/met_win_clearinterval.asp"
  origin: "http://www.w3schools.com"
  pathname: "/jsref/met_win_clearinterval.asp"
  port: ""
  protocol: "http:"
  ► reload: reload()
  ► replace: ()
  search: ""
  ► toString: tostring()
  ► valueOf: valueOf()
  ► __proto__: Location
```

La propriété « location.href » et la méthode « location.reload() » permettent toutes deux de recharger la page courante. Exemples d’utilisation ci-dessous :

```
<input type="button" value="Reload Page"
      onClick="window.location.href=window.location.href">

<input type="button" value="Reload Page" onClick="window.location.reload()">
```

On peut aussi, à partir d’une page donnée, forcer une redirection vers la page d’origine :

```
window.location.href = window.location.origin;
```

Le site W3Schools fournit un tableau de synthèse intéressant sur la propriété « location » :

Property	Description
hash	Sets or returns the anchor part (#) of a URL
host	Sets or returns the hostname and port number of a URL
hostname	Sets or returns the hostname of a URL
href	Sets or returns the entire URL
origin	Returns the protocol, hostname and port number of a URL
pathname	Sets or returns the path name of a URL
port	Sets or returns the port number of a URL
protocol	Sets or returns the protocol of a URL

<u>search</u>	Sets or returns the querystring part of a URL
Method	Description
<u>assign()</u>	Loads a new document
<u>reload()</u>	Reloads the current document
<u>replace()</u>	Replaces the current document with a new one

Parmi les méthodes les plus couramment utilisées de l'objet Window, on trouve « onload() » et « onresize() ». Exemple :

```
window.onload = function() {
    mainModule.init();
    SecondModule.init();
}
window.onresize = function() {
    SecondModule.init();
}
```

On rencontre moins fréquemment la méthode « onerror() », mais elle peut permettre dans certains cas de router les message d'erreur vers une autre destination que la console, par exemple la fonction « alert() » ou une « div » de notification (à utiliser en mode débogage uniquement). Si on souhaite utiliser cette méthode « onerror() », on aura intérêt à la placer au tout début du code Javascript de l'application concernée :

```
window.onerror = function (message, url, lineNo){
    var message = 'Error: ' + message + '\n' +
        url + '\n' + 'Line Number: ' + lineNo ;

    var tempdiv = document.createElement("div");
    var tempmsg = document.createTextNode(message);
    tempdiv.appendChild(tempmsg);
    document.getElementsByTagName('body')[0].appendChild(tempdiv);

    tempdiv.style='width:300px;background:orange;padding:3px;font-size:13px';

    console.log(message);

    return true;
}
```

Petite astuce à noter : la méthode `setInterval()` associée à l'objet « `window` » permet de déclencher l'exécution de tâches à intervalles réguliers. Le petit exemple proposé par le site W3Schools illustre très bien ce principe, avec l'affichage d'une petite horloge que l'on peut stopper par un clic de souris :

http://www.w3schools.com/jsref/met_win_clearinterval.asp

```
<p id="demo"></p>

<button onclick="myStopFunction()">Stop time</button>

<script>
var myVar = setInterval(function(){ myTimer() }, 1000);

function myTimer() {
    var d = new Date();
    var t = d.toLocaleTimeString();
    document.getElementById("demo").innerHTML = t;
}

function myStopFunction() {
    clearInterval(myVar);
}
</script>
```

4.5.2 Compléments (popup, lookup)

L'objet Window peut être utilisé dans une page HTML pour déclencher la fermeture de la page courante, après avoir rechargé la page à l'origine de l'ouverture de la page courante :

```
<script type="text/javascript">
    window.location.href = window.location.origin;
</script>
```

On peut aussi utiliser la fonction « open » associée à l'objet Window pour ouvrir une fenêtre de type « popup ». Par exemple, un clic sur la balise « href » ci-dessous a pour effet d'appeler une fonction calc_open_popup(), dont le rôle consiste à ouvrir une fenêtre de type popup :

```
<a href="#">affichcalc.php" id="link">Calculette en mode "popup"</a>
```

Code source de la fonction calc_open_popup() :

```
<script type="text/javascript">
    var link = document.querySelector('a');
    link.addEventListener('click', calc_open_popup, false);

    function calc_open_popup(evt){
        evt.stopPropagation();
        evt.preventDefault();

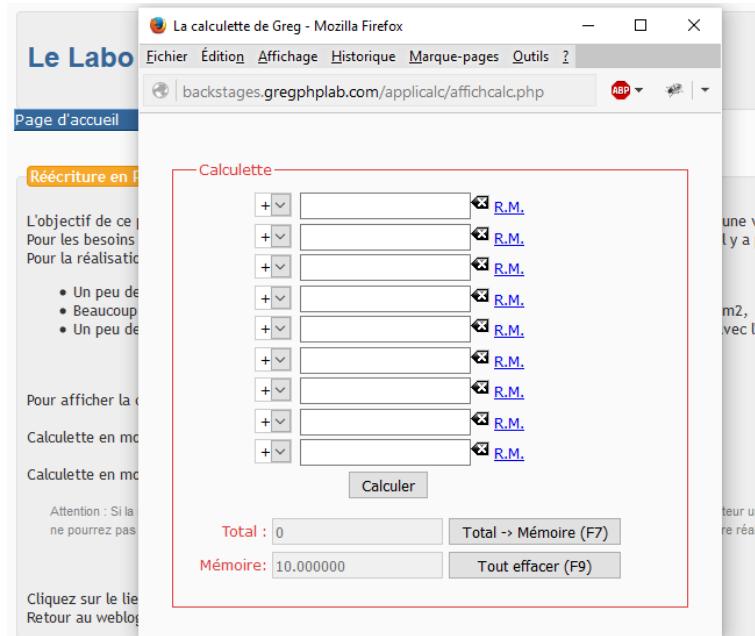
        var url_target = this.getAttribute("href") ;
        /*
         * Il est possible de personnaliser l'affichage de la fenêtre popup
         * au moyen des options ci-dessous
         */
        var popup_options =
'directories=yes,menubar=yes,status=yes,location=yes,scrollbars=no,resizable=no,fullscreen=no,width=500px,height=450px,left=50px,top=50px';

        window.open(url_target, 'Calculette_en_mode_popup',popup_options);
    }
</script>
```

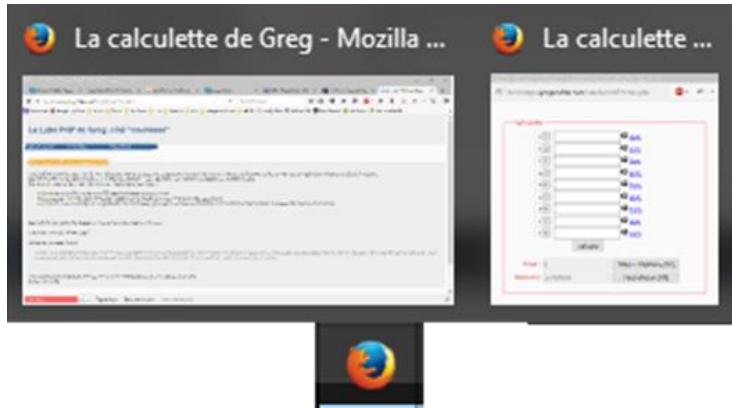
AVERTISSEMENT : si les fenêtres popup ont été beaucoup utilisées dans le passé, une nouvelle tendance est apparue aux alentours de 2010 : les fenêtre modales. Il s'agit de « div » qui viennent se juxtaposer à la page courante, en exploitant une propriété CSS particulière (zindex) qui permet d'afficher certaines informations, comme par exemple le contenu d'une “div”, par dessus les éléments de la fenêtre courante. La fenêtre courante demeure néanmoins active, il n'y a donc pas de risque de perdre le focus

par rapport à une fenêtre particulière. Le projet Bootstrap (de Twitter) propose une implementation de fenêtre modale qui sera abordée en TP.

Dans l'exemple ci-dessous, on voit un exemple de fenêtre popup qui vient se superposer à la fenêtre courante :



Le problème avec les fenêtre popup, c'est que l'utilisateur peut dans certains cas perdre le focus sur la fenêtre en cours, et se trouver obligé de la rechercher dans la liste des fenêtre ouvertes par le navigateur:



Autre astuce intéressante : en supposant que la fenêtre courante s'appelle "popup.html" et que l'on souhaite ouvrir une popup pointant sur le même fichier html, cette popup se trouvera "derrière" la fenêtre courante. Pour contourner cette difficulté, forcer la fermeture de la fenêtre courante, et forcer le focus sur la popup via la fonction focus(), comme dans l'exemple suivant :

```
function OuvrirVisible() {
    var w = window.open("popup.html","pop1","width=200,height=200");
    w.document.close();
    w.focus();
}
```

Il est possible d'afficher une popup pour afficher une liste de selection, dans laquelle on ira "piquer" une donnée renvoyée vers un champ de saisie de la fenêtre appelante. On désigne parfois ce type de popup par le terme de "lookup". Voici un exemple de formulaire avec l'extrait du code source d'une page HTML affichant un formulaire de ce type :

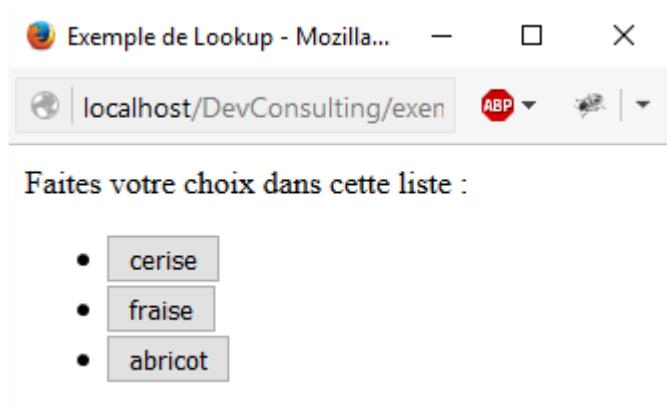
Test de Lookup :

Votre choix

```
<script language="javascript" type="text/JavaScript">
    function LookupOpen(lookup_url,lookup_name,lookup_options) {
        window.open(lookup_url,lookup_name,lookup_options);
    }
</script>

<body>
    <p>Test de Lookup : </p>
    <form name='origine'>
        Votre choix <input type="text" name="choix_val" disabled="disabled" />
        <input type="text" name="choix_lib" disabled="disabled" />
        <input type="button" value="Ouvrir le popup"
onClick="LookupOpen( 'popup.php?form=origine&val=choix_val&lib=choix_lib', 'popupchoix'
,'directories=no,menubar=no,status=no,location=no,scrollbars=no,resizable=yes,height=
500,width=600,left=50,top=50,fullscreen=no')" /><BR>
    </form>
</body>
```

Et voici la fenêtre “popup.php” et son code source :



```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Exemple de Lookup</title>
<script language="javascript" type="text/JavaScript">
    function getLookupRetrieve(val, lib) {
        window.opener.document.forms["origine"].elements["choix_val"].value=val;
        window.opener.document.forms["origine"].elements["choix_lib"].value=lib;
        window.close();
    }
</script>
</head>
<body>
Faites votre choix dans cette liste : <br />
<ul>
<li><input type="button" value="cerise"
           onClick="getLookupRetrieve('1', 'cerise');" /></li>
<li><input type="button" value="fraise"
           onClick="getLookupRetrieve('2', 'fraise');" /></li>
<li><input type="button" value="abricot"
           onClick="getLookupRetrieve('3', 'abricot');" /></li>
</ul>
</body>
</html>
```

On trouvera d'autres exemples de popup sur le site suivant :

<http://www.toutjavascript.com>

4.5.3 L'objet Screen

L'objet Screen est supporté par la plupart des navigateurs, même s'il n'est pas normalisé. Le site W3Schools fournit un tableau de synthèse sur les propriétés fournies par cet objet.

Property	Description
<u>availHeight</u>	Returns the height of the screen (excluding the Windows Taskbar)
<u>availWidth</u>	Returns the width of the screen (excluding the Windows Taskbar)
<u>colorDepth</u>	Returns the bit depth of the color palette for displaying images
<u>height</u>	Returns the total height of the screen
<u>pixelDepth</u>	Returns the color resolution (in bits per pixel) of the screen
<u>width</u>	Returns the total width of the screen

5 Accessibilité (WAI-ARIA)

Avec la montée en puissance de Javascript, on a vu se généraliser l'utilisation de techniques de rafraîchissement de page partiels, notamment au travers d'AJAX. Cette approche permet d'obtenir des applications plus réactives et conviviales, tout en réduisant le volume des échanges entre postes clients et serveurs.

Mais ces techniques posent des problèmes aux technologies d'assistance, lesquelles sont conçues sur la base de l'ancien modèle requête/réponse. L'un des effets de bord les plus visibles est que le bouton "Retour" est souvent inopérant car chaque clic de souris ne déclenche plus nécessairement le chargement d'une nouvelle page. De plus, les logiciels d'assistance à la lecture ne sont pas en mesure de déterminer l'état de telle ou telle portion de la page, et ne sont pas en mesure d'informer l'utilisateur de certains changements intervenus sur la page.

Pour répondre à ce type de problèmes, le W3C travaille sur un projet appelé WAI-ARIA pour "Web Accessibility Initiative's Accessible Rich Internet Applications". Présenté brièvement au début du cours, dans la partie consacrée à la présentation du HTML5, ce projet va être présenté maintenant plus en détail.

<http://www.w3.org/WAI/intro/aria.php>

ARIA propose l'implémentation d'un système de traitement au sein des navigateurs permettant de définir des propriétés, associées aux éléments HTML d'une page, qui indiqueront aux technologies d'assistance le rôle et le statut de ces éléments HTML.

Les travaux du W3C sont très avancés, et la spécification en est au stade de "recommandation final". On peut dès lors et déjà utiliser ses fonctions au sein des applications webs, facilement et sans craindre d'effets de bords sur les principaux navigateurs.

En ce qui concerne les technologies d'assistance, Jaws 7.1, Zoomtext 9, Windows Eyes 5.5 et d'autres prennent déjà ARIA en charge.

Le projet ARIA peut aider les technologies d'assistance dans les cas suivants :

- la définition des rôles dans la structure de la page ;
- une interaction silencieuse en arrière-plan avec le serveur ;
- la navigation par le clavier pour des composants personnalisés (des widgets) ;
- navigation entre états au sein d'une application et d'un widget

Le projet ARIA apporte une série de propriétés, que le développeur associe aux balises existant dans ses pages, attributs qui sont interprétés aussi bien par les navigateurs que par les technologies d'assistance.

Définition des rôles dans la structure de la page

ARIA crée aussi un sous-groupe de propriétés pour les pages et les widgets afin d'aider à préciser leur structure. Ces propriétés sont appelées "*rôles*" et sont classées en 2 catégories :

- les rôles de repères (en anglais "landmark roles"), qui sont des portions de la page destinées à être utilisés comme repères de navigation,
- les rôles de widgets.

Les rôles de repères incluent :

- application
- banner
- complementary
- contentinfo
- form
- main
- navigation
- search

Les rôles de widgets incluent :

- alert
- alertdialog
- button
- checkbox
- combobox
- dialog
- grid
- gridcell
- link
- listbox
- log
- marquee
- menu
- menubar
- menuitem
- menuitemcheckbox
- menuitemradio
- option
- progressbar
- radio
- radiogroup
- scrollbar

- slider
- spinbutton
- status
- tab
- tablist
- tabpanel
- textbox
- timer
- tooltip
- tree
- treegrid
- treeitem

Premier exemple :

```
<div>
    <h3 id="radio_btn">Choisissez votre film favori:</h3>
    <ul class="radiogroup" id="rg1" role="radiogroup" aria-labelledby="radio_btn">
        <li id="r1" role="radio" >Matrix</li>
        <li id="r2" role="radio" >Inception</li>
        <li id="r3" role="radio" >Avalon</li>
        <li id="r4" role="radio" >Blade Runner</li>
        <li id="r5" role="radio" >Le Seigneur des Anneaux</li>
    </ul>
</div>
```

Navigation assistée au sein d'une application et d'un widget

Un composant d'une page fournit des informations intéressantes pour les logiciels d'assistance, notamment une notion de statut (ou d'état). Par exemple, une page peut disposer d'un état "connecté ("Logged") qui chargera certains éléments d'interface utilisateur en fonction de la personne qui sera connectée. De même, un bouton radio peut avoir un état "sélectionné" qui sera lancé lorsque l'utilisateur sélectionnera cet objet.

ARIA propose un groupe d'états et de "propriétés" qui peuvent être utilisés dans divers contextes pour donner des informations précises aux user-agents et aux lecteurs d'écran durant le cycle de vie d'une application. On trouve notamment les propriétés suivantes :

- aria-autocomplete
- aria-checked (state)
- aria-disabled (state)
- aria-expanded (state)
- aria-haspopup
- aria-hidden (state)
- aria-invalid (state)
- aria-label

- aria-level
- aria-multiline
- aria-multiselectable
- aria-orientation
- aria-pressed (state)
- ariareadonly
- aria-required
- aria-selected (state)
- aria-sort
- aria-valuemax
- aria-valuemin
- aria-valuenow
- aria-valuetext

En reprenant l'exemple précédent, vous pouvez utiliser la propriété aria-checked pour désigner l'objet de la liste qui sera sélectionné par défaut.

```
<div>
    <h3 id="radio_btn">Choisissez votre film favori:</h3>
    <ul class="radiogroup" id="rg1" role="radiogroup" aria-labelledby="radio_btn">
        <li id="r1" role="radio" aria-checked="false">Matrix</li>
        <li id="r2" role="radio" aria-checked="false">Inception</li>
        <li id="r3" role="radio" aria-checked="false">Avalon</li>
        <li id="r4" role="radio" aria-checked="true">Blade Runner</li>
        <li id="r5" role="radio" aria-checked="false">Le Seigneur des
            Anneaux</li>
    </ul>
</div>
```

L'élément suivant est par conséquent défini comme étant sélectionné.

```
<li id="r4" role="radio" aria-checked="true">Blade Runner</li>
```

Interaction silencieuse en arrière-plan avec le serveur

Le point le plus complexe à interpréter pour un lecteur d'écran est le contexte dans lequel l'application va charger des données en arrière-plan et mettre à jour les éléments de l'interface utilisateur. ARIA offre un nouveau panel de propriétés qui permettent de déclarer des régions particulières dans une page et qui avertissent le "user-agent" lors d'un changement.

- *aria-live* : Spécifie qu'un élément va être mis à jour et décrit les types de mise à jour auxquelles les navigateurs, les technologies d'assistance et les utilisateurs peuvent s'attendre dans cette portion de page. Elle accepte les valeurs off (la portion n'est pas active), polite (notify les utilisateurs lors d'une mise à jour, mais n'interrompt pas la tâche en cours) et assertive (la mise à jour est communiquée à l'utilisateur de manière urgente).

```
<div id="LiveRegion" role="contentInfo" aria-live="assertive" >
    Cette portion est active.</div>
```

- *aria-atomic*. Indique si les technologies d'assistance vont présenter la portion modifiée en totalité ou en partie. Elle accepte les valeurs true et false.

```
<div id="LiveRegion" role="contentInfo" aria-live="assertive"
      aria-atomic="true" >Cette portion est active.</div>
```

- *aria-relevant*. Indique les changements pertinents au sein d'une portion. Elle accepte les valeurs suivantes :

Valeur	Description
additions	Des noeuds d'éléments sont ajoutés au DOM dans la portion active.
removals	Des noeuds d'éléments ou de texte de la portion active sont retirés du DOM.
text	Du texte est ajouté à n'importe quel noeud DOM enfant de la portion active.
all	Équivaut à la combinaison de toutes les valeurs «addition, removals, text».
additions text (valeur par défaut)	équivaut à la combinaison des valeurs «addition et text».

- *aria-busy*. Indique si un élément est en train d'être mis à jour. Accepte la valeur true ou false.

Navigation par le clavier pour des composants personnalisés

Les utilisateurs se servent du clavier pour naviguer bien plus qu'on ne le pense. Il est donc essentiel d'en tenir compte et aussi de ne pas lier l'application à des événements qui dépendent de périphériques, comme la souris (par exemple l'événement rollover).

Sur ce point, ARIA étend la propriété tabindex qui permet déjà d'assigner un ordre de navigation aux objets présents sur une page avec la touche de tabulation. Cette propriété était déjà prise en charge par le HTML mais uniquement pour les propriétés a, area, button, object, input, select et textarea. Elle est maintenant utilisable sur la plupart des éléments d'une page.

```
<div>
    <h3 id="radio_btn">Choisissez votre film favori:</h3>
    <ul class="radiogroup" id="rg1" role="radiogroup"
        aria-labelledby="radio_btn">
        <li id="r1" tabindex="1" role="radio" aria-checked="false" >Matrix</li>
        <li id="r2" tabindex="2" role="radio" aria-checked="false" >Inception</li>
        <li id="r3" tabindex="3" role="radio" aria-checked="false" >Avalon</li>
        <li id="r4" tabindex="5" role="radio" aria-checked="true" >Blade
            Runner</li>
        <li id="r5" tabindex="4" role="radio" aria-checked="false" >Le Seigneur
            des Anneaux</li>
    </ul>
</div>
```

L'élément ci-dessous est par conséquent défini comme étant sélectionné.

```
<li id="r4" tabindex="5" role="radio" aria-checked="true">Blade Runner</li>
```

Dans l'exemple précédent, nous avons associé un tabindex aux éléments "li" d'un élément "ul", qui simule un groupe de boutons radio.

La propriété tabindex d'ARIA accepte également les valeurs négatives, ce qui permet d'exclure un objet mais de recevoir néanmoins le focus *via* une navigation au clavier.

Résumé

L'accessibilité est un problème de dimension sociale important pour le développement du Web. Lorsqu'une application, ou un site web, est bien conçue et bien développée, tous les utilisateurs ont un accès égal à ses informations et à ses fonctionnalités.

Le HTML5 propose de nouveaux éléments, propriétés et attributs, à l'usage des développeurs afin de pouvoir rendre le contenu plus accessible.

L'annexe du présent support fournit un chapitre contenant plusieurs exemples d'implémentation de WAI-ARIA.

Plusieurs lecteurs d'écran existent pour Windows. Deux sont des produits commerciaux (JAWS et Window-Eyes) et un troisième est gratuit (NVDA) :

- JAWS : http://en.wikipedia.org/wiki/JAWS_%28screen_reader%29
- Window-Eyes : <http://fr.wikipedia.org/wiki/Window-Eyes>
- NVDA : <http://www.nvda-fr.org/>

Des solutions équivalentes existent sur d'autres plateformes :

- Linux : Orca (<http://live.gnome.org/Orca>)
- Mac : VoiceOver (solution intégrée via les Préférences Système → Système → Accès universel)

Un aperçu de la façon de travailler avec Firefox et NVDA est présenté sur le site suivant :

<http://www.marcozehe.de/2009/04/14/article-on-how-to-use-nvda-and-firefox-to-test-web-sites-for-accessibility/>

Le site WebAIM donne un bon aperçu des différentes formes de handicap types :

<http://www.iheni.com/screen-reader-testing>

6 APIs HTML 5

HTML5 est une norme en constante évolution et de nouvelles APIs sont susceptibles d'être intégrées par le W3C, de même que les APIs présentées dans ce support de cours sont susceptibles d'évoluer. Il est donc recommandé de mettre en place une veille technologique, de manière surveiller les évolutions de la norme HTML5.

On notera que, faute de temps, le fonctionnement de plusieurs APIs n'a pas été précisé dans ce support.

6.1 *QuerySelector*

6.1.1 Rappel et compléments

Les APIs *QuerySelector()* et *QuerySelectorAll()*

L'API *QuerySelector* constitue une avancée majeure du HTML 5.

On rappelle qu'avant la norme HTML5, pour sélectionner des éléments du DOM, on ne disposait que de ceci :

```
document.getElementById('foo');
document.getElementsByClassName('bar');
document.getElementsByTagName('p');
```

Quelques exemples :

```
<script>
var items = document.getElementsByTagName("li");
for (var i=0; i < items.length; i++) {
    console.log(typeof items[i]);
}

var sections = document.getElementsByTagName("article");
for (var i = 0; i < sections.length; i++ ) {
    if (sections[i].getAttribute("id") != id) {
        sections[i].style.display = "none";
    } else {
        sections[i].style.display = "block";
    }
}

if (document.getElementById(sectionId)) {
```

```
        document.getElementById(sectionId).style.display = "none";
    }
</script>
```

C'était pas mal, mais un peu faible pour parcourir le DOM et sélectionner précisément des éléments à manipuler.

L'API `QuerySelector()`.

Avec l'introduction de l'objet "classList", Javascript a rendu la manipulation des classes (et pseudo-classes) plus facile.

Comme son nom l'indique, l'objet "classList" est un regroupement de toutes les classes qui sont rattachées à un élément du DOM.

Exemple avec la sélection du premier élément associé à la classe "foo" :

```
var el = document.querySelector('.foo');
console.log(el.classList);
```

L'objet "classList" résultant de la sélection contient une série de propriétés et méthodes facilitant la manipulation de classes HTML.

On peut tout d'abord vérifier la présence d'une classe sur un élément via la méthode `contains()`. On peut également ajouter une classes avec la méthode `add()`, et en supprimer avec la méthode `remove()`.

Exemples :

```
if (el.classList.contains('foo')) {
    el.classList.remove('foo');
} else {
    el.classList.add('foo');
}
```

La norme ECMAScript 5 offre un raccourci avec la méthode `toggle()`. Le code ci-dessous est donc strictement équivalent au code qui précède :

```
el.classList.toggle('foo');
```

Rappelons également que la méthode `QuerySelector()` permet de sélectionner tout type d'attribut, comme dans les exemples suivants :

- Exemple de sélection d'un attribut "role" ayant pour valeur "main" :

```
<div role="main">Lorem ipsum...</div>

<script>
var el = document.querySelector("[role='main']");
console.log(el) ; // <div role="main">
</script>
```

- Exemple de sélection d'un attribut "data-prix" ayant pour valeur 200 :

```
<script>
var el = document.querySelector("[data-prix=200]");
</script>
```

Voici un exemple de tableau pas très respectueux de la norme HTML (absence de quelques balises "fermantes"), et voyons comment en extraire des informations selon 2 méthodes :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<style>
table>tbody>tr:nth-child(even) {
    background-color: yellow;
}
table>tbody>tr:nth-child(odd) {
    background-color: silver;
}
</style>
<body>
<table id="score">
    <thead>
        <tr>
            <th>Produit
            <th>Résultat
    <tfoot>
        <tr>
            <th>Moyenne
            <td>82%
    <tbody>
        <tr>
            <td>Produit A
            <td>87%
        <tr>
            <td>Produit B
            <td>78%
        <tr>
            <td>Produit C
            <td>81%
    </tbody>
</table>
```

```

<script>
/*
 * Méthode 1 : à l'ancienne
 */
var table = document.getElementById("score");
var groups = table.tBodies;
var rows = null;
var cells = [];

for (var i = 0, imax = groups.length; i < imax; i++) {
  rows = groups[i].rows;
  for (var j = 0, max = rows.length; j < jmax; j++) {
    cells.push(rows[j].cells[1]);
  }
}
console.log(cells); // on obtient un tableau : [td, td, td]

/*
 * Méthode 2 : avec querySelectorAll, en une seule ligne
 */
var cells2 = document.querySelectorAll("#score>tbody>tr>td:nth-of-type(2)");
console.log(cells2); // on obtient une NodeList[td, td, td]

</script>

</body>
</html>

```

On constate aisément le gain qu'apporte querySelectorAll() en termes de nombre de lignes de code :

```
var cells2 = document.querySelectorAll("#score>tbody>tr>td:nth-of-type(2));
```

Dans l'exemple suivant, la variable "vlinks" contient une liste de liens que le visiteur a visité durant sa navigation sur la page.

will acquire a list of links that the user has visited. The author can then obtain the URLs and potentially exploit this knowledge.

```

var vlinks = document.querySelectorAll(":visited");
for (var i = 0; i < vlinks.length; i++) {
  doSomethingEvil(vlinks[i].href);
}

```

Autre exemple :

```
<div id="foo">
```

```

<p class="warning">This is a sample warning</p>
<p class="error">This is a sample error</p>
</div>
<div id="bar">
  <p>...</p>
</div>

<script>
var alerts = document.querySelectorAll("p.warning, p.error");
console.log(alerts); // NodeList[p.warning, p.error]

var x = document.querySelector("#foo, #bar");
console.log(x); // <div id="foo">

var x = document.querySelector("#bar, #foo");
console.log(x); // <div id="foo">

</script>

```

La méthode `querySelector()` peut aussi être utilisée sur des éléments sélectionnés via un évènement :

```

function handle(evt) {
  var x = evt.target.querySelector("span");
  ...
  // Do something with x
}

```

La méthode `querySelector()` peut aussi être utilisée sur un élément préalablement "filtré" via une méthode comme `getElementById()` :

```

var div = document.getElementById("bar");
var p = div.querySelector("body p");

```

Exemple de "parcours" des éléments d'une `NodeList` :

```

<ul class="nav">
  <li><a href="/">Accueil</a></li>
  <li><a href="/produits">Produits</a></li>
  <li><a href="/apropos">A propos</a></li>
</ul>

<script>
var lis = document.querySelectorAll("ul.nav>li");
for ( var i = 0; i < lis.length; i++) {
/*
   Adressage d'élément avec la méthode item() et le numéro
   d'indice du tableau "lis"
*/

```

```

        console.log(lis.item(i).firstChild);

        /*
            Adressage d'élément utilisant la syntaxe propre aux tableaux
            (pour un résultat identique)
        */
        console.log(lis[i].firstChild);
    }
</script>

```

En reprenant l'exemple précédent, voici un exemple de manipulation du DOM, avec la suppression des balises "<a>" se trouvant dans le menu :

```

<ul class="nav">
    <li><a href="/">Accueil</a></li>
    <li><a href="/produits">Produits</a></li>
    <li><a href="/apropos">A propos</a></li>
</ul>

<script>
    function process(elmt) {
        /*
            * Rappel: il est nécessaire de "remonter" sur le parent de
            * l'élément à supprimer avant de supprimer l'élément
        */
        elmt.parentNode.removeChild(elmt);
    }

    var lis = document.querySelectorAll("ul.nav>li");
    for ( var i = 0; i < lis.length; i++) {
        /*
            * Suppression de chaque élément de "lis"
        */
        process(lis.item(i).firstChild);

    }
</script>

```

IMPORTANT : L'exemple ci-dessus démontre un atout de la méthode `querySelectorAll()` : cette dernière génère une NodeList qui n'est pas "live", c'est à dire que son contenu est statique, et pas du tout impacté par les manipulations ultérieures opérées sur le DOM. Dans une NodeList "live", la suppression de chacun des éléments de la NodeList aurait entraîné une réindexation de la NodeList, avec pour effet de bousculer la perte des indexs définissant chaque élément.

Autres exemples :

```

<script>
// sélectionner tous les paragraphes
var allParagraphs = document.querySelectorAll('p');

```

```
// sélectionner tous les paragraphes qui sont des "descendants" directs d'un élément "p".
var myClassParagraphs = document.querySelectorAll('p > span.myClass') ;

</script>
```

Autre exemple pour jouer avec les sélecteurs CSS :

```
<ul>
<li>Hello1</li>
<li>Hello2</li>
<li>Hello3</li>
<li>Hello4</li>
<li id='Last'>Hello5</li>
</ul>
<script>
console.log(document.querySelector('li').textContent) ; // Hello1
console.log(document.getElementById('last').textContent) ; // Hello5
console.log(document.querySelector('ul>li:nth-of-type(4)').textContent) ; // Hello4
console.log(document.querySelector('ul>li:last-child').textContent) ; // Hello5
</script>
```

Un autre exemple pour rappeler que derrière chaque élément sélectionné se cache un objet :

```
<a href="#">Salut!!</a>
<!-- On rappelle que la balise &lt;a&gt; est un objet héritant de HTMLAnchorElement --&gt;
&lt;script&gt;
    // on prend une référence sur la balise &lt;a&gt;
    var nodeAnchor = document.querySelector('a');
    // création d'un tableau pour stocker les clés
    //   des propriété de l'élément sélectionné
    var props = [];
    // boucle sur l'élément pour en extraire toutes les propriétés
    for ( var key in nodeAnchor) {
        props.push(key);
    }
    console.log(props.sort());
        // nombre de propriétés trop important pour être affiché ici
    console.log(document.querySelector('a').nodeType);
        // logs 1 ce qui correspond à la valeur de Node.ELEMENT_NODE
    console.log(document.querySelector('a').nodeValue); // logs "null"
    console.log(document.querySelector('a').nodeName);
        // logs A ce qui correspond à la balise &lt;a&gt;
    console.log(document.querySelector('a').firstChild.nodeType);
        // logs 3 ce qui correspond à Node.TEXT_NODE
    console.log(document.querySelector('a').firstChild.nodeValue); // logs Salut!!
    console.log(document.querySelector('a').firstChild.nodeName); // logs #text
&lt;/script&gt;</pre>

```

Dans le cas particulier de document contenant des éléments appartenant à plusieurs namespaces : l'API querySelectorAll() ne supporte pas la notion de namespace, mais il est possible de contourner le problème en utilisant la technique suivante :

```
<svg id="svg1" xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xLink">
  <video id="svgvideo1" xlink:href="myvideo.ogg" width="320" height="240"/>
  <foreignObject width="100" height="100">
    <video id="htmlvideo1" src="myvideo.ogg" xmlns="http://www.w3.org/1999/xhtml">No
video1</video>
  </foreignObject>
</svg>
<script>
  var elms = document.querySelectorAll("svg video");
  var result = new Array();
  var svgns = "http://www.w3.org/2000/svg"
  for ( var i = 0; i < elms.length; i++) {
    if (elms[i].namespaceURI == svgns) {
      result.push(elms[i]);
    }
  }
</script>
```

6.1.2 Sommer une colonne de tableau

Le code ci-dessous présente un exemple de modification dynamique du contenu d'un tableau HTML, modification consistant à sommer les montants de la seconde colonne, et à ajouter une ligne "total" dans le pied du tableau.

Dans cet exemple, 2 solutions (au moins) permettent de sélectionner uniquement la seconde colonne du tableau :

- solution 1

```
var cells = dataTable.querySelectorAll("td + td"); // NodeList[td, td, td]
```

- solution 2

```
var cells = dataTable.querySelectorAll("#table1>tbody>tr>td:nth-of-type(2)"); // NodeList[td, td, td]
```

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<style>
#table1>tbody>tr:nth-child(even) {
    background-color: yellow;
}
#table1>tbody>tr:nth-child(odd) {
    background-color: silver;
}
</style>
<body>

<script type="text/javascript">
//<![CDATA[
window.onload = function() {
    var sum = 0;
    // On prend un raccourci sur l'élément "tableau"
    var dataTable = document.getElementById("table1");

    /*
        Sélectionner la 2ème colonne du tableau pour en extraire les
        montants (2 techniques équivalentes)
        var cells = dataTable.querySelectorAll("td + td");
        => NodeList[td, td, td]
    */
    var cells = dataTable.querySelectorAll(
        "#table1>tbody>tr>td:nth-of-type(2)"); // NodeList[td, td, td]
    console.log(cells);

    // C'est le moment d'attaquer les calculs
    var i, i_length = cells.length;
    for (i = 0; i < i_length; i++) {
```

```

        sum += parseFloat(cells[i].firstChild.data);
    }

    // Ajout d'une somme en fin de tableau
    var newRow = document.createElement("tr");
    // Création de la 1ère cellule de la ligne
    var firstCell = document.createElement("td");

    var firstCellText = document.createTextNode("Somme : ");
    firstCell.appendChild(firstCellText);
    newRow.appendChild(firstCell);

    // Création de la 2ème cellule de la ligne (contenant le cumul)
    var secondCell = document.createElement("td");
    var secondCellText = document.createTextNode(sum);
    secondCell.appendChild(secondCellText);
    newRow.appendChild(secondCell);
    newRow.style.backgroundColor = "#ff0000";
    // Ajout de la ligne en fin de tableau
    dataTable.appendChild(newRow);
}

//--><!]]>
</script>

<table id="table1">
    <tr>
        <td>Produit A</td>
        <td>145</td>
    </tr>
    <tr>
        <td>Produit B</td>
        <td>233</td>
    </tr>
    <tr>
        <td>Produit C</td>
        <td>833</td>
    </tr>
</table>

</body>
</html>
```

On en profite pour rappeler les opérations permettant d'ajouter dynamiquement un élément dans une page. En l'occurrence, il s'agit ici d'ajouter une ligne à un tableau :

1. créer une nouvelle ligne en utilisant `document.createElement("tr")`.
2. créer chaque cellule de la nouvelle ligne en utilisant `document.createElement("td")`.
3. créer pour chaque nouvelle cellule du contenu textuel en utilisant `document.createTextNode()`
4. Attacher (Append) le noeud textuel à chaque cellule du tableau
5. Attacher les nouvelles cellules à la nouvelle ligne

6. Attacher la nouvelle ligne au tableau

6.1.3 Compléments

L'annexe du cours propose 2 documents qui sont les suivants :

- un tableau récapitulant les différents sélecteurs CSS disponibles
- une série d'exercices pour renforcer la compréhension du système des sélecteurs

6.2 LocalStorage et SessionStorage

Les API HTML5 Local Storage et Session Storage fournissent un moyen pour les sites web de stocker des informations sur le poste client, informations qui seront récupérables ultérieurement.

Le concept est assez proche de celui des cookies, mais il est conçu pour le stockage de données de taille plus importantes. On rappelle que les cookies sont limités en taille, et surtout que le navigateur les renvoie à chaque requête HTTP.

Les données stockées en local via l'API Session Storage sont perdues dès la fermeture du navigateur.

A l'inverse, les données stockées via l'API Local Storage sont conservées après fermeture du navigateur, et elles seront de nouveau accessibles si l'utilisateur se reconnecte sur le même sous-domaine.

Les données stockées localement via l'API Local Storage, sont librement accessibles via Javascript.

Comment contrôler si un navigateur supporte les API Local Storage et Session Storage :

Si votre navigateur supporte l'API Local Storage, alors l'objet "window" doit contenir une propriété "localStorage" (idem pour sessionStorage).

Si votre navigateur ne supporte pas le stockage HTML5, la valeur de la propriété localStorage sera "undefined" (idem pour sessionStorage).

Exemple de fonction permettant de déterminer la présence de l'API Local Storage :

```
function supports_local_storage() {
    try {
        return 'localStorage' in window && window['localStorage'] !== null;
    } catch(e) {
        return false;
    }
}
```

On peut dupliquer le code ci-dessus pour l'API sessionStorage.

Exemple de test équivalent sous Modernizr :

```
if (Modernizr.localstorage) {
    // window.localStorage est disponible
} else {
    // pas de support natif pour localstorage
}
```

Pour l'API sessionStorage, on utilisera le raccourci Modernizr.sessionstorage.

On notera que l'attribut localStorage sous Modernizr est intégralement en minuscules, alors que la propriété équivalente du DOM est appelée window.localStorage (S de Storage en majuscules).

Exemple d'utilisation pour sessionStorage :

```
// 3 manières strictement équivalentes de mettre à jour un élément
sessionStorage.setItem('cours1','SQL DB2 pour développeurs et
administrateurs');
sessionStorage['cours2'] = 'PHP Avancé';
sessionStorage.cours3 = 'Javascript et HTML5';

var cours1 = sessionStorage.getItem('cours1');
console.log(cours1) ; // SQL DB2 pour ...
var cours2 = sessionStorage['cours2'];
console.log(cours2) ; // PHP Avancé
var cours3 = sessionStorage.cours3;
console.log(cours3) ; // Javascript et HTML5
```

On peut adapter l'exemple à l'API localStorage, les techniques d'écriture étant strictement les mêmes.

On peut stocker jusqu'à 5 Mo de données pour chaque sous-domaine . C'est en tout cas le chiffre recommandé dans la spécification HTML5.

On peut supprimer un élément de stockage en utilisant la méthode removeItem() qui, comme getItem(), prend un nom de clé unique comme argument et supprime l'élément correspondant :

```
sessionStorage.removeItem ( "cours1" ) ;
```

On peut supprimer en une seule instruction tous les éléments stockés dans sessionStorage, ou dans localStorage, via la méthode clear() :

```
sessionStorage.clear ();
localStorage.clear ();
```

En théorie, il devrait être possible d'affecter un écouteur d'évènement "storage" à l'unité de stockage localStorage. Cet évènement renverrait un objet avec des propriétés telles que la clé qui a changé, ainsi que l'ancienne et la nouvelle valeur associées à cette clé :

```
window.addEventListener('storage', function (e) {
    var msg = 'Key ' + e.key + ' changed from ' + e.oldValue + ' to ' +
e.newValue;
    console.log(msg);
}, false);
```

Dans les faits, il semble que cette fonctionnalité ne soit pas encore implémentée dans Firefox et Chrome.

Pour rappel, les données stockées dans sessionStorage et localStorage peuvent être affichés

dans l'onglet "Ressources" de l'outil de développement de Google Chrome :

Key	Value
adValue	<style contenteditable> #ad { background-color: #fff; width: 300px; height: 250px; }</style>
field1	sdfsqd
field2	qsdffqsd
field3	eeeeee
field4	aeraezrrzae

Exemple de script implémentant l'API localStorage :

```
<!DOCTYPE html>
<head>
<meta charset="utf-8">
<style type="text/css">
  * {
    -webkit-tap-highlight-color: rgba(0,0,0,0);
    outline: none;
    text-rendering: optimizeLegibility;
  }
  #cta {
    font: 175% sans-serif;
    text-align: center;
    margin-bottom: 20px;
  }
  #ad {
    display: block;
    text-align: center;
  }
  #ad:hover {
    background-color: #999;
    width: 300px;
    height: 250px;
  }
  button {
    margin-top: 20px;
    margin-bottom: 20px;
    width: 300px;
    height: 30px;
  }
</style>
```

```
<body align="center">
  <div id="cta">Edit Your Own CSS!</div>
    <div id="ad" contenteditable>
      <div id="style">
        <style contenteditable>
          #ad {
            background-color: #fff;
            width: 300px;
            height: 250px;
            border: 1px solid #000;
          }
        </style>
      </div>
    </div>
    <button id="clearValues">clear storage</button>
    <div id="output"></output>

    <script type="text/javascript">
      var adStyle = document.querySelector('#style'),
          clearIt = document.querySelector('#clearValues'),
          output = document.querySelector('#output');

      function adInit () {
        if (localStorage.getItem('adValue') === 'null' ||
            localStorage.getItem('adValue') === null) {
          console.log('init')
        } else {
          adStyle.textContent = localStorage.getItem('adValue');
          console.log(localStorage.getItem('adValue'))
          output.textContent = "Values Loaded!!!";
        }

        adStyle.addEventListener('DOMCharacterDataModified', updateAdStyle,
          false); //Fires everytime a character is changed
        clearIt.addEventListener('click', clear, false);

        adStyle.focus();
      }

      function updateAdStyle () {
        if(localStorage) {
          output.textContent = "Values Saved!!!";
          console.log(adStyle.textContent);
          //store the values
          var styleFix = "<style contenteditable>" + adStyle.textContent +
                         "</style>";
          localStorage.setItem('adValue', styleFix);
        }
      }

      function clear () {
        if(localStorage.getItem('adValue') != 'null' ||

```

```
        localStorage.getItem('adValue') != null) {
            localStorage.clear();
        }
        output.textContent = "";
        console.log('clear')
    }

    window.addEventListener('DOMContentLoaded', adInit, false);

</script>
</body>
</html>
```

6.3 Geolocation

Pour contrôler la disponibilité de l'API Geolocation avec Modernizr :

```
if (Modernizr.geolocation) {
    // let's find out where you are!
} else {
    // no native geolocation support available :(
    // maybe try Gears or another third-party solution
}
```

Exemple trouvé dans le livre "HTML 5 Advertising"

```
<!DOCTYPE html>
<head>
<meta charset=utf-8>
<body>
<header>
    <h1>geolocation</h1>
    <div id="coords"></div>
</header>
<script>
function success(position) {
    var lat = position.coords.latitude;
    var long = position.coords.longitude;
    document.getElementById('coords').innerHTML = "<p><strong>lat: </strong>" + lat + "<br><strong>long: </strong>" + long + "";
}

function error(error) {
    switch (error.code) {
        case error.PERMISSION_DENIED:
            alert("user did not share geolocation data");
            break;
        case error.POSITION_UNAVAILABLE:
            alert("could not detect current position");
            break;
        case error.TIMEOUT:
            alert("retrieving position timed out");
            break;
        default:
            alert("unknown error");
            break;
    }
}

function adInit(event) {
    console.log(event.type)
    if (navigator.geolocation) {
```

```
navigator.geolocation.getCurrentPosition(success, error);
} else {
  error('not supported');
}
}

window.addEventListener('DOMContentLoaded', adInit, false);
</script>
</body>
</html>
```

Tutoriel sur W3Schools :

https://www.w3schools.com/html/html5_geolocation.asp

6.4 Drag & Drop

L'API « Drag and Drop » du HTML5 est une API relativement facile à implémenter, qui permet d'effectuer du « drag and drop » sans avoir besoin de recourir à un framework particulier, dès lors que le navigateur supporte cette API (ce qui est le cas de la plupart des navigateurs récents).

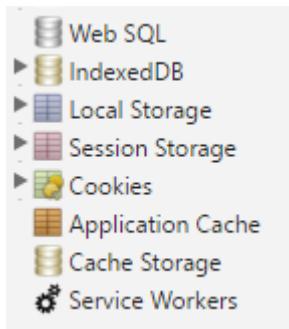
On trouvera sur internet de nombreux tutoriaux présentant le fonctionnement de cette API. On recommandera pour débuter d'étudier l'exemple fourni sur W3Schools :

http://www.w3schools.com/HTML/html5_draganddrop.asp

Un chapitre a été prévu en annexe (chapitre 9.x) expliquant comment effectuer un « drag and drop » entre 2 listes HTML.

6.5 IndexedDB et WebSQL

Un débat semble-t-il houleux s'est déroulé au sein du W3C sur le besoin de créer une API de type base de données (SQL ou pas) au sein des navigateurs. Pour résumer, il y avait les supporters de la solution SQL (essentiellement Apple) et donc de l'API WebSQL. Et de l'autre côté, les supporters de la solution no-SQL et donc de l'API IndexedDB. C'est finalement IndexedDB qui l'a emporté, et c'est la seule des 2 API qui a été normalisée par le W3C, l'autre ayant été abandonnée. Ceci étant, certains navigateurs implémentent quand même WebSQL, comme Google Chrome (cf. les ressources visibles dans les outils de développement du navigateur :



Il semble que cette décision du W3C ait déplu aux ingénieurs d'Apple, qui ont implémenté WebSQL dans Safari, et une version demeurée boguée de IndexedDB dans ce même navigateur.

On trouve de nombreuses ressources sur internet expliquant le fonctionnement d'IndexedDB :

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB

L'utilisation d'IndexedDB est un peu plus complexe que celle d'API comme LocalStorage, elle ne sera pas détaillée dans ce chapitre.

On notera que plusieurs initiatives visant à proposer une implémentation de SQL en Javascript se sont développées en 2015, dont les projets suivants :

<https://github.com/google/lovefield/>

<http://alasql.org>

<http://jsperf.com/sql-js-vs-alasql-js/4>

<http://kripken.github.io/sql.js/>

On peut sans doute s'attendre à voir d'autres initiatives de ce genre se développer courant 2016. On notera par exemple que le projet Lovefield utilise IndexedDB comme support pour le

stockage des données côté navigateur, sauf sur Safari où là le projet utilise WebSQL.

Exemple de projet très intéressant implémentant une couche ORM (Object Relational Mapping) sur IndexedDB :

<https://github.com/pavelpat/indexeddb-example>

6.6 Canvas

Canvas est une API dédiée à la génération de graphiques et d'animations. Comme SVG, elle s'appuie sur un moteur de rendu vectoriel, mais produit une sortie de type bitmap.

Avec Canvas, une fois que le graphique est dessiné, le navigateur l'oublie. Si des éléments du canvas doivent être modifiés, c'est toute la scène qui doit être redessinée.

Cette API étant riche en fonctionnalités, il conviendrait de lui consacrer un livre entier. Il existe d'ailleurs de très bonnes références, et en particulier les deux livres suivants, disponibles chez l'éditeur Apress.com (collection FriendsOf) :

- Foundation HTML5 Animation with JavaScript, de Keith Peters et Billy Lamberta (2011)
- Foundation Game Design with HTML5 and JavaScript, de Rex Van der Spuy (2012)

La chaîne Youtube CodingMaths, animée par Keith Peters, est parfaite pour progresser dans le domaine de l'animation avec Canvas (tous les exemples présentés par Keith Peters sont écrits en Vanilla JS) :

https://www.youtube.com/channel/UCF6F8LdCSWIRwQm_hfA2bcQ

Le site W3Schools propose un bon tutoriel pour s'initier à Canvas :

https://www.w3schools.com/html/html5_canvas.asp

Nicolas Legrand, formateur spécialiste de Canvas, a rédigé un excellent support qui permet de s'initier à Canvas, et qui explique aussi comment contourner certaines limites de l'API :

[http://the-tiny-spark.com/books/creer-un-moteur-d'affichage-2d-en-HTML5.pdf](http://the-tiny-spark.com/books/creer-un-moteur-d-affichage-2d-en-HTML5.pdf)

Quelques projets et frameworks s'appuyant sur Canvas :

<http://p5js.org/>

<https://github.com/thetinyspark/tomahawk>

<http://fabricjs.com/>

<http://paperjs.org/>

<http://createjs.com/getting-started/easeljs>

<http://ocanvas.org/>

Quelques frameworks spécialisés dans le développement de jeux :

<https://phaser.io/>

<http://craftyjs.com/>

Moteurs de rendu d'effets physique 2D :

<https://github.com/scheppe/p2.js>

<http://brm.io/matter-js/>

Quelques bons tutoriaux :

https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial

<http://www.html5canvastutorials.com/>

<https://developer.mozilla.org/fr/docs/Web/HTML/Canvas>

<https://www.alsacreations.com/tuto/lire/1484-introduction.html>

Canvas permet d'afficher aussi du graphisme 3D via l'API WebGL :

https://developer.mozilla.org/fr/docs/Web/API/WebGL_API/Tutorial/Commencer_avec_WebGL

<https://www.khronos.org/webgl/>

<https://threejs.org/>

<https://www.alsacreations.com/tuto/lire/1572-webgl-3d-three-canvas-threejs.html>

<https://codepen.io/msfeldstein/project/editor/XLryLD/>

Exemple de Canvas simple (image statique) emprunté au site W3Schools :

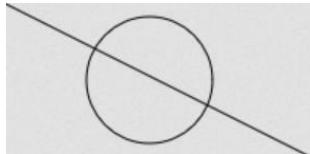
```
<!DOCTYPE html>
<html>
<body>

<canvas id="myCanvas" width="200" height="100" style="border:1px solid #d3d3d3;">
    Your browser does not support the HTML5 canvas tag.</canvas>

<script>
    var c = document.getElementById("myCanvas");
    var ctx = c.getContext("2d");
    ctx.moveTo(0,0);
    ctx.lineTo(200,100);
    ctx.stroke();
    ctx.beginPath();
    ctx.arc(95,50,40,0,2*Math.PI);
    ctx.stroke();
</script>

</body>
</html>
```

Résultat obtenu :



On notera qu'il est possible de créer un Canvas à la volée, en pur JS, il faut simplement penser à l'attacher à un élément du DOM, comme par exemple le nœud « body » :

```
var canvas = document.createElement('canvas');
var target = document.querySelector('body');
target.appendChild(canvas);
```

Pour bénéficier de la meilleure fluidité possible sur des Canvas animés, il est recommandé d'utiliser l'API « requestAnimationFrame », accessible via l'objet JS de même nom qui est associé à l'objet « window ». Un chapitre est consacré à la présentation de cette API, au sein de ce support.

Dans l'exemple d'horloge animée proposée par W3Schools sur cette page :

https://www.w3schools.com/graphics/tryit.asp?filename=trycanvas_clock_start

... l'animation des aiguilles est réalisée par la fonction « drawClock » qui est lancée en boucle, à raison d'une fois par seconde, via la fonction JS « setInterval » :

```
setInterval(drawClock, 1000);
```

Cette méthode classique fonctionne bien, mais pour garantir une fluidité optimale, on peut remplacer la ligne ci-dessus par la ligne suivante :

```
requestAnimationFrame(drawClock);
```

Pour que cela fonctionne correctement, il faut aussi ajouter ce même appel à la fin de la fonction « drawClock » :

```
function drawClock() {
    drawFace(ctx, radius);
    drawNumbers(ctx, radius);
    drawTime(ctx, radius);
    requestAnimationFrame(drawClock); // ligne ajoutée
}
```

L'API « requestAnimationFrame » est privilégiée par les développeurs, pour le développement de jeux et pour toutes animations nécessitant une très bonne fluidité.

On trouvera en annexe un exemple de création d'une forme géométrique bien connue, l'épicycloïde. Au-delà de l'aspect ludique, cet exemple est intéressant car il combine un certain nombre de techniques, telles que :

- l'utilisation des attributs data-*,
- l'utilisation de l'objet « requestAnimationFrame » (présenté page précédente),
- la génération à la volée de plusieurs Canvas dans une même page,
- la possibilité de générer une même forme en tant qu'image fixe ou animée

On trouvera enfin, dans le chapitre consacré à SVG, un tableau comparant les avantages et inconvénients de SVG et Canvas.

Un chapitre entier de ce support est consacré à la console et aux outils de développement. Parmi les outils de développement proposés par Firefox, il en est un qui est particulièrement intéressant pour analyser le processus de tracé d'un Canvas. En cliquant plusieurs fois sur l'icône  pendant qu'un Canvas est en train de se générer, on obtient une série de clichés que l'on peut analyser après coup :



Le cliché ci-dessus a été pris à partir du tracé d'une forme géométrique appelée « rose mystique » ou « napperon ». On peut obtenir des clichés similaires à partir du tracé d'épicycloïde proposé en annexe.

On peut développer des effets de parallaxe avec Canvas. L'article suivant, de Chris Hadlock, constitue une excellente introduction au sujet :

<https://www.ibm.com/developerworks/library/wa-parallaxprocessing/>

L'exemple de Chris Hadlock utilisait un peu de JQuery, je l'ai réécrit en VanillaJS, vous pouvez voir le résultat ici (et récupérer facilement le code source) :

http://backstages.gregphlab.com/games/canvas_parallax/index.html

Vous pouvez aussi voir une adaptation de Pong (et récupérer facilement son code source) ici :

<http://backstages.gregphlab.com/games/pong/index.html>

6.7 SVG

SVG signifie « Scalable Vector Graphics ».

SVG existait bien avant l'arrivée de HTML5. En 2011, le W3C a annoncé l'adoption de SVG comme recommandation pour le tracé de dessin vectoriel en HTML. Avant cette annonce du W3C, le support de SVG dans les navigateurs n'était pas homogène, et on devait utiliser un plugin spécifique sur certains navigateurs pour afficher du SVG, ce qui était peu pratique.

La reconnaissance de SVG par le W3C a constitué un formidable accélérateur pour SVG, qui est aujourd'hui supporté par la plupart des navigateurs, notamment sur les mobiles. Les fichiers SVG étant peu volumineux, et donc peu gourmands en bande passante, ils sont très prisés des designers pour l'habillage et l'animation d'icônes sur les sites mobiles.

W3Schools constitue une bonne ressource pour démarrer l'étude de SVG :

https://www.w3schools.com/html/html5_svg.asp

https://www.w3schools.com/graphics/svg_intro.asp

On peut légitimement s'interroger sur les avantages de SVG par rapport à Canvas. Les deux solutions présentent des avantages et inconvénients, listés dans ce tableau proposé par W3Schools :

Canvas	SVG
<ul style="list-style-type: none"> • Dépendant de la résolution • Pas de support de gestionnaire d'évènements • Possibilités de rendu du texte limitées • Possibilité de sauvegarder les images en .png et .jpg • Bien adapté pour le développement de jeux utilisant du graphisme de manière intensive (jeu de plateformes, RPG, etc..) 	<ul style="list-style-type: none"> • Indépendant de la résolution • Support du gestionnaire d'évènements (du navigateur) • Bien adapté pour le rendu de graphiques et de cartes (Google Maps) • Relativement lent (dépendant du DOM) • Peu adapté au développement de jeux

SVG s'appuie sur XML, et chaque graphique SVG intégré à une page web fait partie intégrante du DOM (lui et ses éléments constitutifs sont reconnus comme des noeuds du DOM). A contrario, dans le cas d'un Canvas, le Canvas lui-même est vu comme un noeud du DOM par le navigateur, mais tous les graphiques générés à l'intérieur du Canvas sont invisibles pour le DOM. C'est d'ailleurs la raison pour laquelle il est impossible de mettre en place des écouteurs d'évènements sur ces éléments, et que l'on est obligé de développer son propre moteur 2D par-dessus Canvas lorsque l'on a besoin de ce type de fonctionnalité (cf. le projet Tomahawk).

Exemple de graphique SVG intégré dans une page HTML :

```
<!DOCTYPE html>
<html>
<body>

<h1>My first SVG</h1>

<svg width="100" height="100">
    <circle cx="50" cy="50" r="40" stroke="green" stroke-width="4"
           fill="yellow" />
    Sorry, your browser does not support inline SVG.
</svg>

</body>
</html>
```

Résultat obtenu :

My first SVG



6.8 Webaudio

L'API Webaudio permet de bénéficier d'un véritable studio d'enregistrement à l'intérieur des navigateurs.

Quelques références permettant de s'initier au sujet :

<https://github.com/alemangui/web-audio-resources>

<https://github.com/alemangui/pizzicato>

<http://alemangui.github.io/meetups/web-audio-pizzicato.pdf>

<https://alemangui.github.io/funk-machine/>

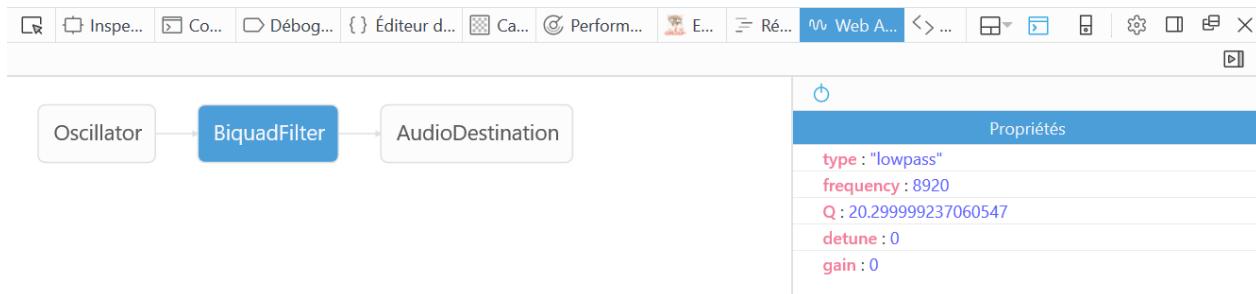
Newsletter de Chris Lowis dédiée au son sur ordinateur :

<http://www.webaudioweekly.com/>

Un très bon cours de Matthew Yee King (de l'Université Goldsmith de Londres), sur l'utilisation de Webaudio :

<https://www.futurelearn.com/courses/electronic-music-tools/>

Certains navigateurs proposent des outils de développement spécifiques à une API. Firefox propose pour Webaudio un graphe permettant de visualiser les différents éléments en place dans un sketch audio :



Voici le code source du sketch audio correspondant au graphe précédent :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Webaudio Labo</title>
</head>
<body>

<div style="background:black; height:500px"
    tabindex="1"
    onkeydown="setNote(event.key);"
    onmousemove = "changeFilter(event.clientX, event.clientY);"
>
</div>

<script>
    var context = window.AudioContext || window.webkitAudioContext;
    var con = new context(); // définition du contexte audio
    var osc = con.createOscillator();

    // low pass filter
    var filter = con.createBiquadFilter();

    // branchement du filtre sur l'oscillateur
    osc.connect(filter);

    // connexion du filtre à la sortie (output)
    filter.connect(con.destination);

    // réglage de la fréquence
    osc.frequency.value = 600;
    filter.frequency.value = 100;

    osc.type = "sawtooth";

    // démarrage de l'oscillateur
    osc.start(0);

    // arrêt de l'oscillateur
    //osc.stop();

    function changeFilter(mx, my) {
        filter.frequency.value = mx * 10;
        filter.Q.value = my / 10;
    }
    // déclenchement de certains sons via le clavier
    function setNote(key) {
        if (key == 'w') {
            osc.frequency.value = 261.63;
        }
        if (key == 'x') {
            osc.frequency.value = 293.66;
        }
        if (key == 'c') {
            osc.frequency.value = 323.63;
        }
        if (key == 'v') {
            osc.frequency.value = 348.23;
        }
    }
</script>
</body>
</html>
```

6.9 Promise

Le fonctionnement de cette API n'est pas détaillé dans cette version du support.

Articles :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise
<http://stackoverflow.com/questions/22516959/how-to-determine-if-a-promise-is-supported-by-the-browser>
https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise
<https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html>
<https://zestedesavoir.com/tutoriels/446/les-promesses-en-javascript/>
<https://developers.google.com/web/fundamentals/getting-started/primers/async-functions>

Exemple :

```
/***
 * Définition d'une promise déclenchant un appel JSONRPC
 * @returns {Promise}
 */
var _promiseStatVente = function () {
    return new Promise(function (resolve, reject) {
        console.time("stat_vente");
        var tmpstatus = false;
        var response = {};
        jsonrpc.request('Dashboard\\Stats:StatVente', {})
            .then(function (result) {
                response.stats_data = result;
                response.stats_year = new Date().getFullYear();
                resolve(response);
            })
            .catch(function (error) {
                reject(response);
            })
            .finally(function () {
                console.timeEnd("stat_vente");
            });
    });
}

// Exécution de la promise (dans un contexte AngularJS 1.6)
_promiseStatVente().then(function (content) {
    $scope.statvte_data = content.stats_data;
    $scope.current_year = content.stats_year;
}).catch(function (err) {
    console.info('Stat Vente KO !');
});
```

6.10 Webworkers

Le fonctionnement de cette API n'est pas détaillé dans cette version du support.

Articles :

http://www.w3schools.com/HTML/html5_webworkers.asp
https://developer.mozilla.org/fr/docs/Utilisation_des_web_workers

Exemple d'utilisation de l'API WebWorkers dans la documentation du framework Math.js :

<http://mathjs.org/examples/browser/webworkers/index.html>

6.11 Websockets

Le fonctionnement de cette API n'est pas détaillé dans cette version du support.

Articles :

<http://www.html5rocks.com/en/tutorials/websockets/basics/>

6.12 Application cache

Le fonctionnement de cette API n'est pas détaillé dans cette version du support.

Articles :

http://www.w3schools.com/HTML/html5_app_cache.asp

6.13 Deviceorientation et Devicemotion

Les API Deviceorientation et Devicemotion sont très importantes pour le développement d'applications mobiles.

Leur fonctionnement n'est pas détaillé dans cette version du support.

Articles :

https://developer.mozilla.org/fr/docs/Web/API/Detecting_device_orientation
<https://www.sitepoint.com/using-device-orientation-html5/>
<https://mobiliforge.com/design-development/html5-mobile-web-device-orientation-events>

<https://www.alsacreations.com/tuto/lire/1501-api-device-orientation-motion-acceleration.html>

<https://www.html5rocks.com/en/tutorials/device/orientation/>

6.14 Touch

Le fonctionnement de cette API est présenté brièvement dans le chapitre 4.5.5.

6.14 RequestAnimationFrame

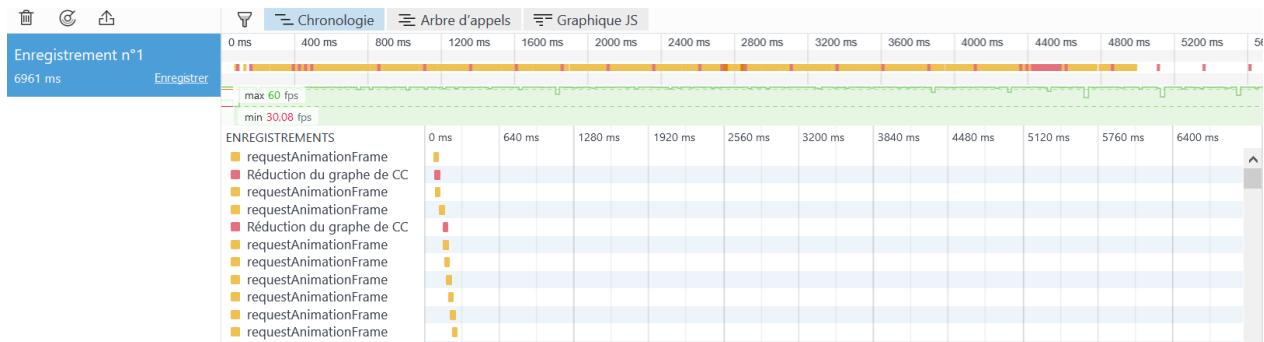
Pour bénéficier de la meilleure fluidité possible sur des Canvas animés, il est nécessaire de passer par l'API « requestAnimationFrame » et l'objet de même nom associé à l'objet « window ».

Pour une présentation détaillée de cette API :

<https://developer.mozilla.org/fr/docs/Web/API/Window/requestAnimationFrame>

[https://msdn.microsoft.com/fr-fr/library/hh920765\(v=vs.85\).aspx](https://msdn.microsoft.com/fr-fr/library/hh920765(v=vs.85).aspx)

Firefox propose dans ses outils de développement, un outil permettant d'analyser le travail réalisé par l'API « requestAnimationFrame ».



Lors de l'évènement « BestOfWeb 2016 », Freddy Harris avait fait une présentation remarquée sur les problématiques de performance liées à l'animation d'une page web. Cette présentation est visualisable ici :

<https://www.youtube.com/watch?v=qrbBD-1ET14>

L'article suivant est un résumé « offline » de la présentation de Freddy Harris :

<http://www.ux-republic.com/web-animation-performance-2/>

A l'époque pas si lointaine où Canvas était supporté expérimentalement par les navigateurs, certains navigateurs utilisaient un préfixe sur l'objet « requestAnimationFrame » (par exemple « moz ») pour

Mozilla Firefox). Pour garantir une compatibilité maximale sur les navigateurs anciens ou pas à jour, il est recommandé d'utiliser un polyfill. Il en existe plusieurs versions, en voici une qui fonctionne bien :

```
// http://paulirish.com/2011/requestAnimationFrame-for-smart-animation/
// http://my.opera.com/emoller/blog/2011/12/20/requestAnimationFrame-for-smart-er-animating

// requestAnimationFrame polyfill by Erik Möller
// fixes from Paul Irish and Tino Zijdel

(function() {
    var lastTime = 0;
    var vendors = ['ms', 'moz', 'webkit', 'o'];
    for(var x = 0; x < vendors.length && !window.requestAnimationFrame; ++x) {
        window.requestAnimationFrame = window[vendors[x] + 'RequestAnimationFrame'];
        window.cancelAnimationFrame = window[vendors[x] + 'CancelAnimationFrame']
            || window[vendors[x] + 'RequestCancelAnimationFrame'];
    }

    if (!window.requestAnimationFrame || !window.cancelAnimationFrame)
        //current Chrome (16) supports request but not cancel
        window.requestAnimationFrame = function(callback, element) {
            var currTime = new Date().getTime();
            var timeToCall = Math.max(0, 16 - (currTime - lastTime));
            var id = window.setTimeout(function() { callback(currTime + timeToCall); },
                timeToCall);
            lastTime = currTime + timeToCall;
            return id;
        };

    if (!window.cancelAnimationFrame)
        window.cancelAnimationFrame = function(id) {
            clearTimeout(id);
        };
})();
```

6.14 Web Animations

L'API Web Animations est apparue en 2012, pour répondre à certaines problématiques d'animation au sein des pages web. La spécification correspondante au W3C est encore à l'état « draft » en mars 2017.

Le fonctionnement de cette API n'est pas détaillé dans cette version du support.

Articles :

<https://pawelgrzybek.com/intro-to-the-web-animations-api/>
<https://css-tricks.com/comparison-animation-technologies/>
<https://w3c.github.io/web-animations/>

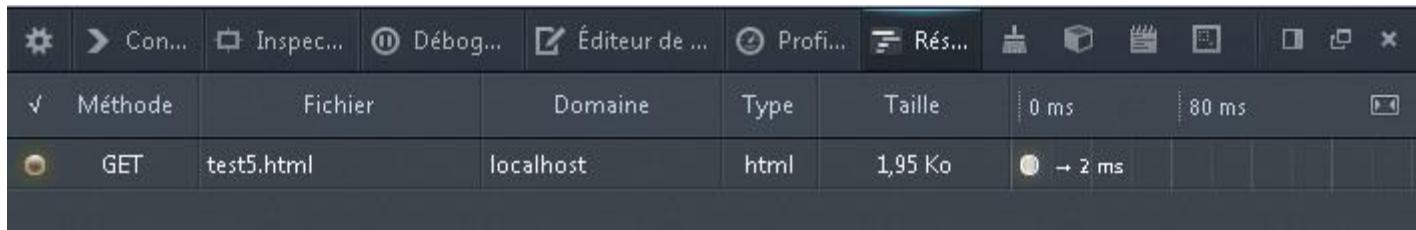
7 Console, et autres outils

Le développeur Javascript est un développeur heureux... car il dispose d'une palette d'outils impressionnante, quel que soit son navigateur de préférence. Sous Windows et Linux, avec Chrome et Firefox (ainsi qu'avec Microsoft IE et Edge), l'accès aux outils de développement se fait via le raccourci F12. Sur les autres navigateurs et systèmes d'exploitation, prière de vous reporter à la documentation correspondante en cas de doute.

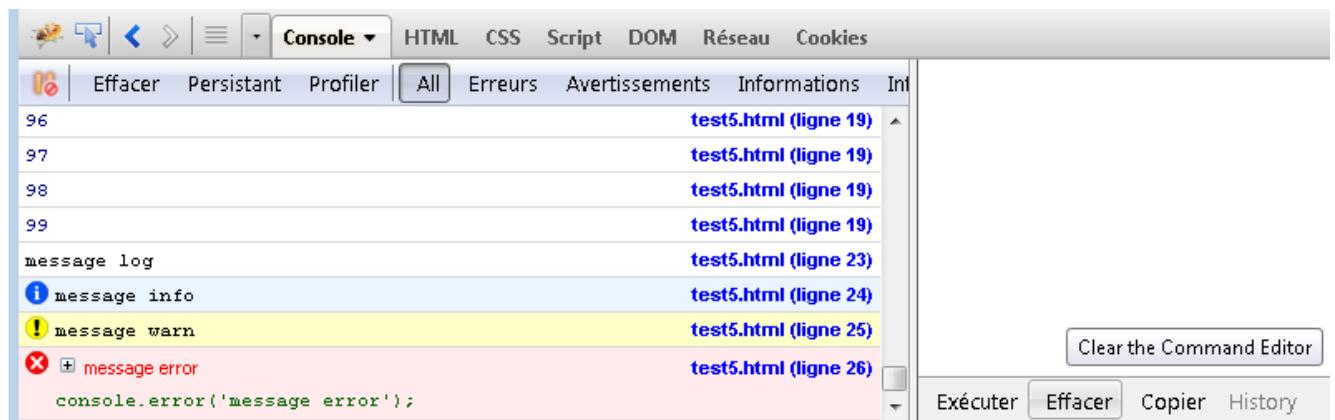
L'autre moyen couramment pratiqué pour accéder aux outils de développement consiste à passer par l'inspecteur d'élément. On accède à cette fonctionnalité en effectuant un clic droit sur n'importe quel élément de la page, et en choisissant l'option « inspecter l'élément ».

Firefox

Le navigateur Firefox fournit en standard un "inspecteur d'élément" de bonne facture, qui permet d'analyser précisément chacun des éléments d'une page (et d'accéder par la même occasion à l'ensemble des outils de développement) :



Beaucoup de développeurs préfèrent à l'inspecteur natif l'usage du plugin Firebug (à installer via l'option "modules complémentaires"), activable via la touche F12 :

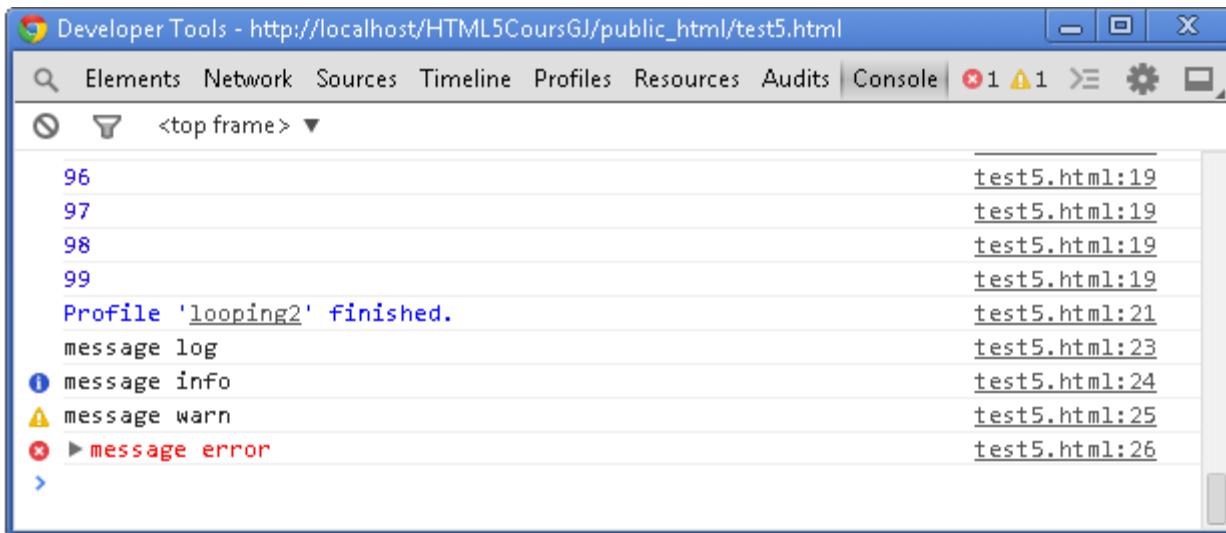


Nous allons beaucoup parler de l'objet « console » dans la suite de ce chapitre, car c'est un formidable outil d'assistance au développement. Cet objet est associé à l'objet « window » et est disponible sur l'ensemble des navigateurs.

Cet objet « console » permet d'envoyer à la console du navigateur des messages d'information, d'avertissement ou d'erreur. Si la méthode « log » de l'objet « console » est probablement la plus utilisée, d'autres méthodes existent qui permettent de qualifier le niveau de gravité du message envoyé :

```
console.log('message log');
console.info('message info');
console.warn('message warn');
console.error('message error');
```

Voici la manière dont ces messages apparaissent dans la console de Google Chrome :



L'objet console fournit aussi la méthode dir() qui se comporte différemment selon qu'on l'utilise sous Firefox ou Chrome :

```
var test = {key1:'valeur 1', key2:'valeur 2'} ;
console.log(test);
console.dir(test);
```

Comportements comparés de console.log et console.dir sur Firefox :

```
>> var test = {key1:'valeur 1', key2:'valeur 2'} ; console.log(test); console.dir(test);
Object { key1: "valeur 1", key2: "valeur 2" }
{key1: "valeur 1", key2: "valeur 2"}
  ↳ key1 : "valeur 1"
    ↳ key2 : "valeur 2"
  ↳ __proto__ : Object
```

Comportements comparés de console.log et console.dir sur Chrome :

```
> var test = {key1:'valeur 1', key2:'valeur 2'} ;
  console.log(test);
  console.dir(test);

▼ Object {key1: "valeur 1", key2: "valeur 2"} ⓘ
  ↳ key1: "valeur 1"
    ↳ key2: "valeur 2"
  ↳ __proto__ : Object

▼ Object ⓘ
  ↳ key1: "valeur 1"
    ↳ key2: "valeur 2"
  ↳ __proto__ : Object
```

On voit que sous Chrome, console.log permet de « déplier » l'arborescence d'un objet, de la même façon qu'avec console.dir.

On peut mesurer le temps d'exécution d'un groupe d'instructions en utilisant les fonctions console.time() et console.timeEnd(). Attention à bien passer la même chaîne aux 2 fonctions pour démarrer et stopper un timer :

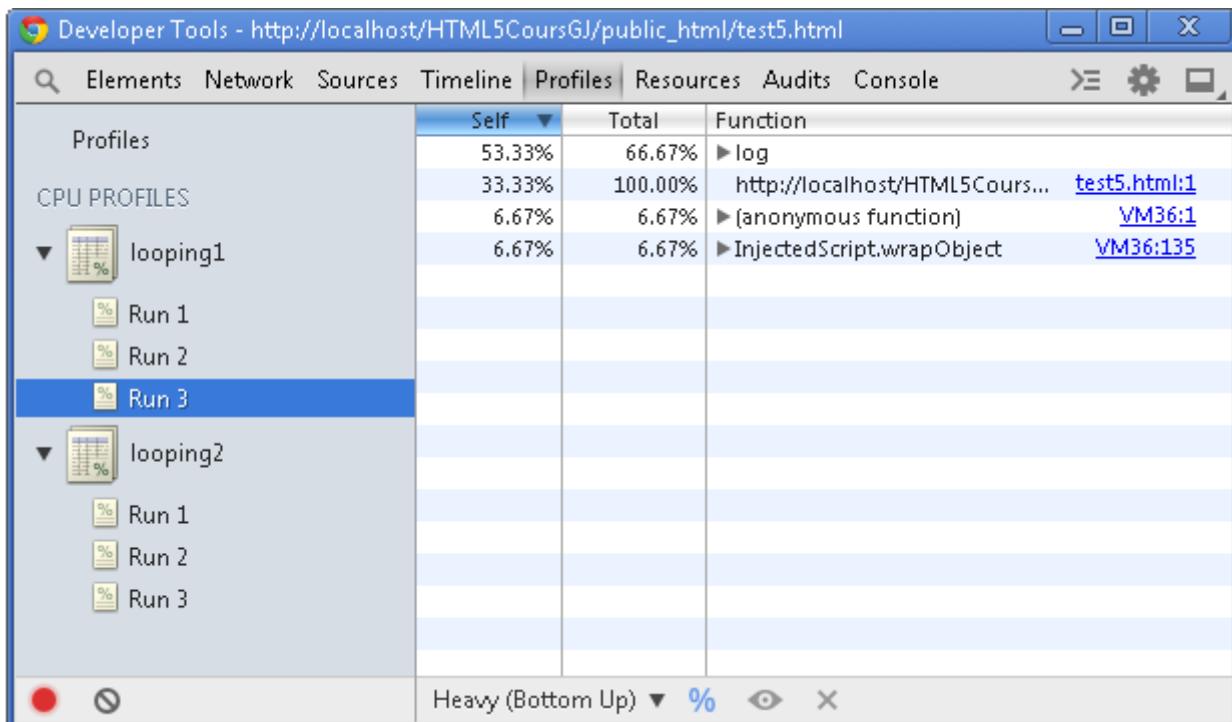
```
console.time("timer1");
for(var i=0 ; i<100 ; i++) {
  console.log(i);
}
console.timeEnd("timer1");
```

On peut profiler plus finement les actions d'un groupe d'instructions via les fonctions `console.profile()` et `console.profileEnd()`

```
console.profile("looping1");
for(var i=0 ; i<100 ; i++) {
    console.log(i);
}
console.profileEnd("looping1");

console.profile("looping2");
for(var i=0 ; i<100 ; i++) {
    console.log(i);
}
console.profileEnd("looping2");
```

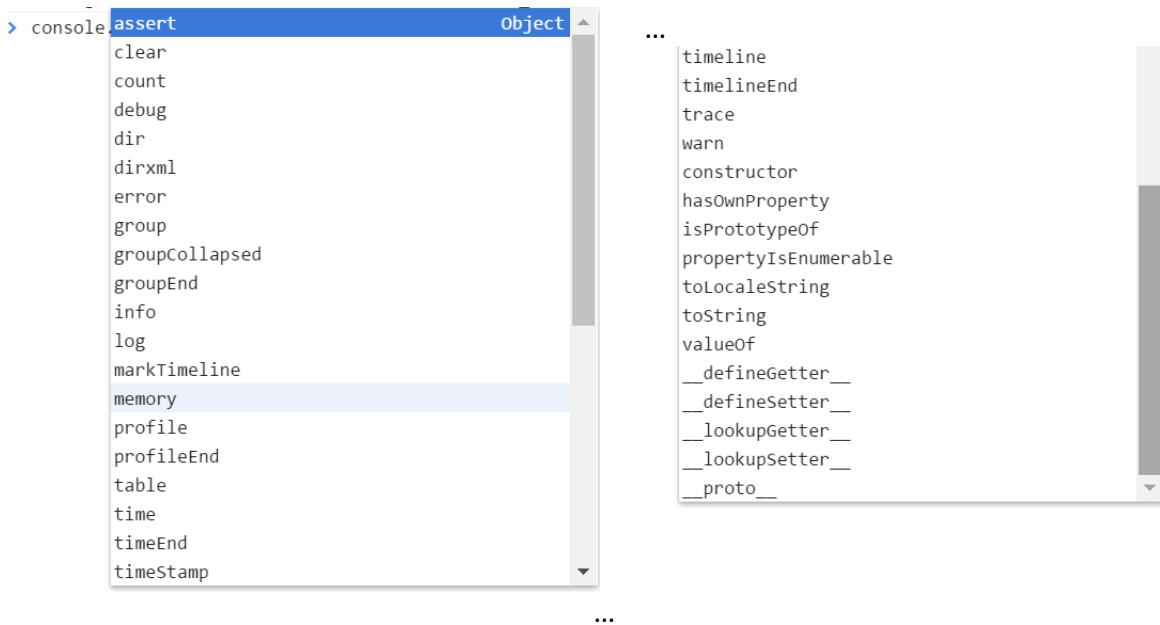
C'est dans Google Chrome que cette fonctionnalité est la plus facile à exploiter :



Il peut être utile, notamment en phase de débogage, de s'assurer de la liste des éléments "script" qui ont été chargés à l'intérieur d'une page. Pour ce faire, on peut recourir au code suivant :

```
console.log(document.querySelectorAll('script')) ;
```

En plus des méthodes déjà évoquées, l'objet « console » offre un certain nombre de méthodes moins connues. L'autocomplétion en ligne de commande permet d'en obtenir un aperçu :



La méthode « table » de l'objet « console » est très pratique pour afficher des objets ayant une structure complexe :

```
var myArray=[{a:1,b:2,c:3},{a:1,b:2,c:3,d:4},{k:11,f:22},{a:1,b:2,c:3}];  
console.table(myArray);
```

```
> var myArray=[{a:1,b:2,c:3},{a:1,b:2,c:3,d:4},{k:11,f:22},{a:1,b:2,c:3}]  
< undefined  
> console.table(myArray);
```

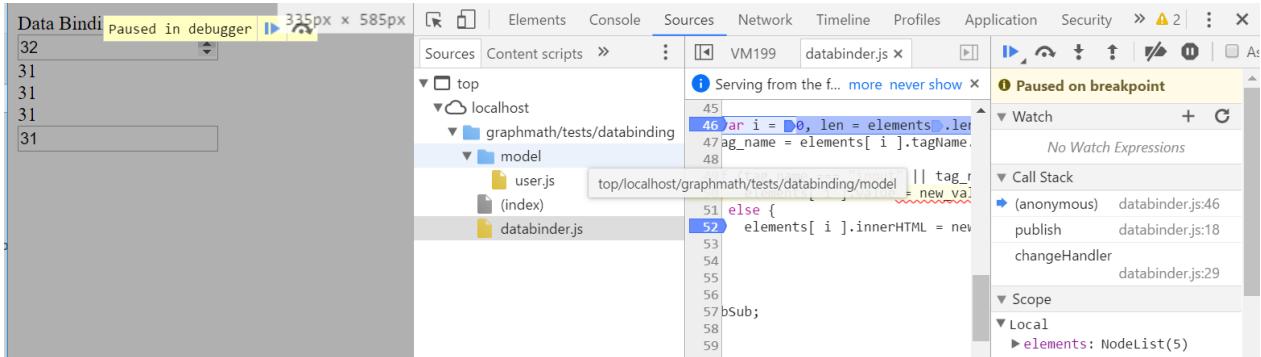
VM481:1

(index)	a	b	c	d	k	f
0	1	2	3			
1	1	2	3	4		
2					11	22
3	1	2	3			

▶ Array(4)

Astuces complémentaires :

- Chrome et Firefox fournissent tous deux la fonction "debugger" que l'on peut lancer dans la console pour forcer le navigateur à se mettre en mode débogage



- Chrome fournit une fonction « monitorEvents » qui permet de moniturer tous les évènements associés à un élément du DOM. Dans l'exemple ci-dessous, on a placé provisoirement un « monitorEvents » sur l'objet « window » avant de le retirer, via la fonction « unmonitorEvents ».

```
> monitorEvents(window)
<- undefined
  devicemotion
    DeviceMotionEvent {acceleration: DeviceAcceleration,
  ▶ accelerationIncludingGravity: DeviceAcceleration, rotationRate:
    DeviceRotationRate, interval: 0, type: "devicemotion"...}
  deviceorientation
    DeviceOrientationEvent {alpha: null, beta: null, gamma: null, absolute:
      false, type: "deviceorientation"...}
> unmonitorEvents(window)
<- undefined
> |
```

Cela fonctionne également avec des éléments sélectionnés de cette manière :

```
var test = document.querySelector('input');
monitorEvents(test);
```

On peut aussi filtrer le ou les évènements à surveiller :

```
monitorEvents(test, 'change');
monitorEvents(test, ['change', 'click']);
```

- Chrome fournit la fonction « getEventListeners » qui renvoie un tableau répertoriant les écouteurs d'évènements associés à un élément du DOM. Par exemple, appliquée à l'objet « window », cela donne :

```
> getEventListeners(window);
<- ▼ Object {load: Array(1)} ⓘ
  ▼ load: Array(1)
    ▼ 0: Object
      ► listener: function () ...
      once: false
      passive: false
      ► remove: function remove()
      type: "load"
      useCapture: false
      ► __proto__: Object
      length: 1
      ► __proto__: Array(0)
      ► __proto__: Object
```

On peut obtenir un niveau de détail plus fin

```
> getEventListeners(window).load[0]
<- ▼ Object {useCapture: false, passive: false, once: false, type: "load", listener: function...}
  ⓘ
  ► listener: function () ...
  once: false
  passive: false
  ► remove: function remove()
  type: "load"
  useCapture: false
  ► __proto__: Object
```

... ou encore :

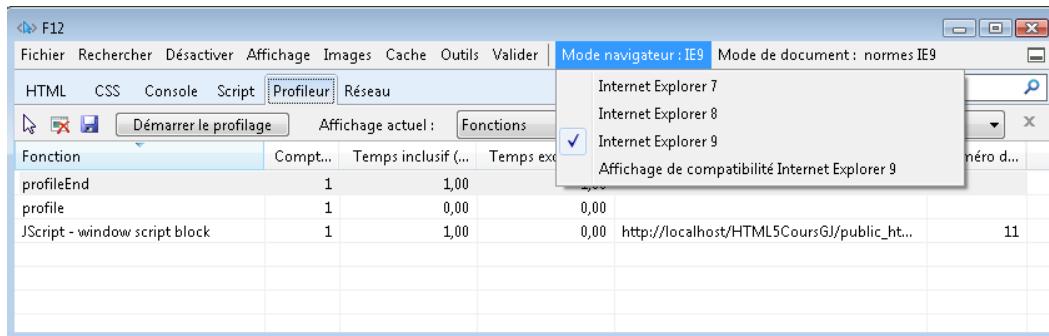
```
> getEventListeners(window).load[0].listener
<- function () {
  var user = new User(123);
  user.set("name", "Wolfgang");
}
```

- On peut vider la console du navigateur de tous ses messages en utilisant la commande :
`clear()`

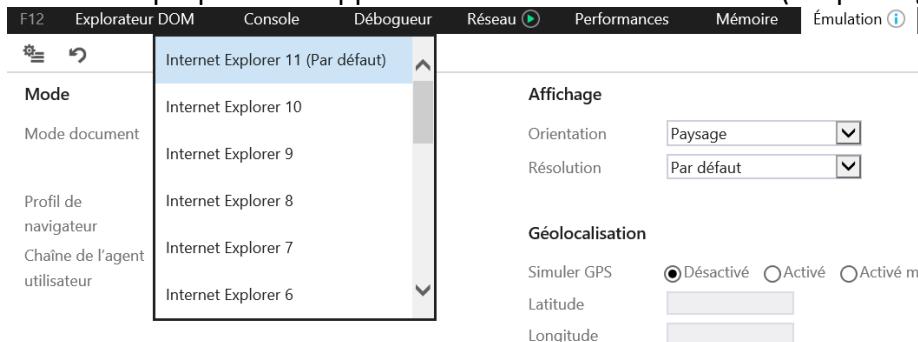
Internet Explorer (IE)

IE fournit peu ou prou les mêmes fonctionnalités que Firebug et Chrome au niveau des outils de développement.

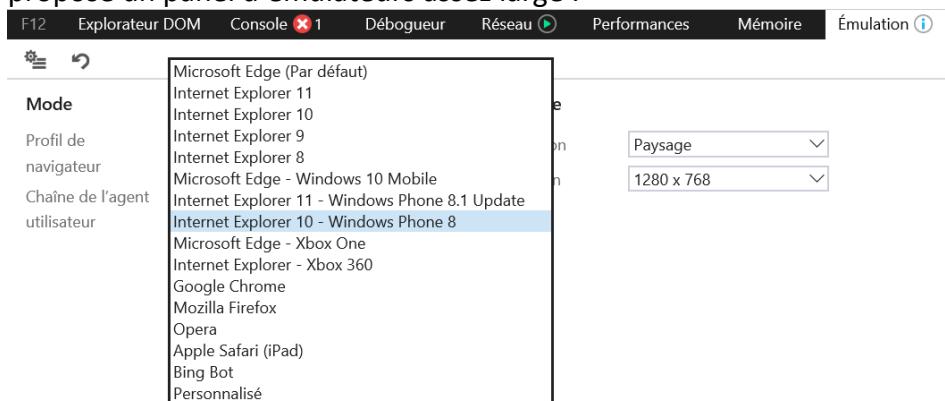
On notera que , mais on soulignera la possibilité de sélectionner un "mode navigateur" pour vérifier la manière dont une page se "comporte" sur IE8 et IE7 :



Microsoft Edge et IE 11 proposent à peu près les même outils de développement, mais IE11 continue à proposer le support de l'émulation de IE6 et IE7 (ce que Edge ne propose pas) :



Le support de ces très vieux navigateurs devient heureusement anecdotique (en 2017). Edge lui propose un panel d'émulateurs assez large :



Google Chrome

L'outil de développement de Google Chrome propose quelques fonctionnalités intéressantes, comme par exemple un affichage simplifié des "ressources", ou espaces de stockage, qui sont spécifiques à certaines API HTML5 (IndexedDB, LocalStorage, SessionStorage, ApplicationCache):

The screenshot shows the Google Chrome Developer Tools with the 'Resources' tab selected. On the left, a tree view shows 'LocalStorage' expanded, with 'http://localhost' selected. The main table lists key-value pairs for this storage:

	Key	Value
field1	sdfsqd	
field2	qsdffqsdf	
field3	eeeeee	
field4	aeraezrrzae	
fieldmultiline		
fieldselect	0	
http://localhost	true	

Les navigateurs basés sur Webkit comme Google Chrome et Safari proposent une option particulièrement intéressante : la "déminification" de code source Javascript. Par exemple, le code source "minifié" de jQuery ci-dessous peut être "déminifié" en cliquant sur le symbole {} situé en bas à gauche de la fenêtre ci-dessous :

The screenshot shows the Google Chrome Developer Tools with the 'Sources' tab selected. A file named 'jquery.min.js' is open. The left pane shows the minified code:

```

1 /*! jQuery v1.10.1 | (c) 2005, 2013 jQuery Foundation, Inc. | jquery.or
2 // @ sourceMappingURL=jquery-1.10.1.min.map
3 */
4 (function(e,t){var n,r,i=typeof t,o=e.location,a=e.document,s=a.documen
5 ),n=s;l=u=r=o=null,t}());var B=/(?:\{[\s\S]*\}|\[([\s\S]*\])$/;P=/([A-
6 u[o]&&(delete u[o],c?delete n[1]:typeof n.removeAttribute!=i?n.remove
7

```

The right pane shows developer tools controls and monitoring information.

Code source "déminifié" :

```

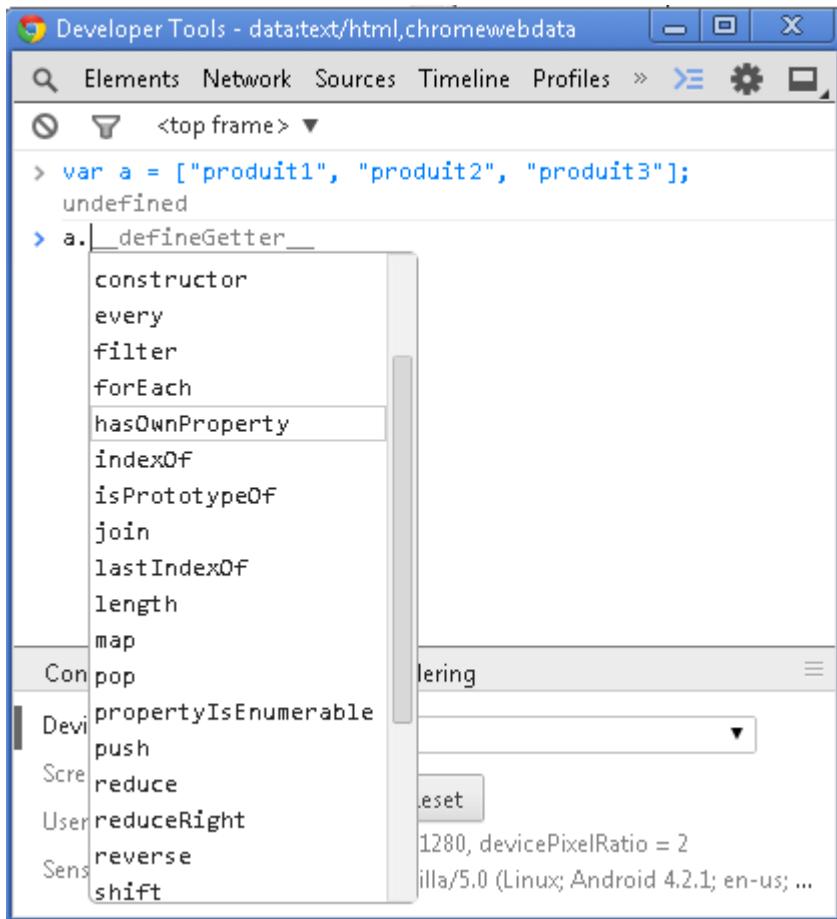
1  /*! jQuery v1.10.1 | (c) 2005, 2013 jQuery Foundation, Inc. | jquery.
2 // @ sourceMappingURL=jquery-1.10.1.min.map
3 */
4 (function(e, t) {
5     var n, r, i = typeof t, o = e.location, a = e.document, s = a.docu
6         return new x.fn.init(e, t, r)
7     }, w = /[+-]?(?:\d*\.|)\d+(?:[eE][+-]\d+|)/.source, T = /\S+/g, C
8         return t.toUpperCase()
9     }, q = function(e) {
10        (a.addEventListener || "load" === e.type || "complete" === a.r
11    }, _ = function() {
12        a.addEventListener ? (a.removeEventListener("DOMContentLoaded"
13    );
14    x.fn = x.prototype = {jquery: f, constructor: x, init: function(e, r
15        var i, o;
16        if (!e)
17            return this;
18        if ("string" == typeof e) {
19            if (i = "<" === e.charAt(0) && ">" === e.charAt(e.leng
20                return !n || n.jquery ? (n || r).find(e) : this.cc
21            if (i[1]) {
22                if (n = n instanceof x ? n[0] : n, x.merge(this, x
23                    for (i in n)
24                        xisFunction(this[i]) ? this[i](n[i]) : th
25                return this
26

```

Points à noter :

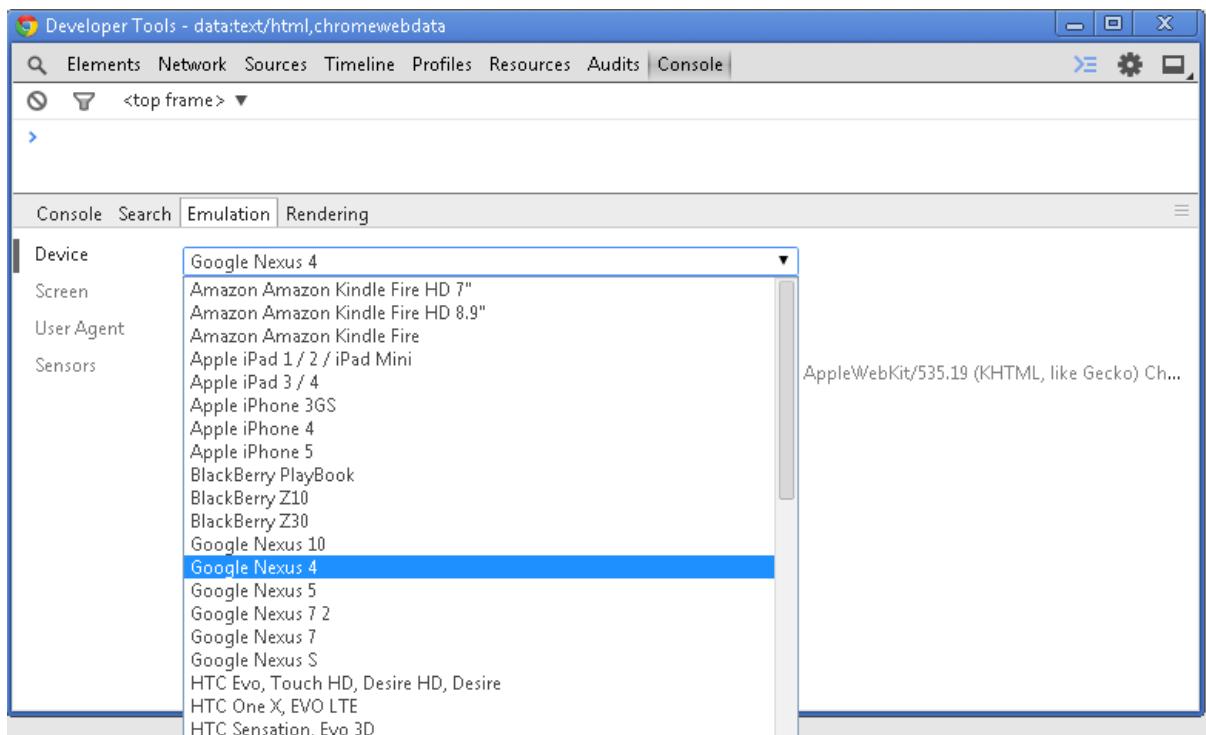
Google Chrome recèle quelques pépites, telles que :

- une auto-complétion automatique et très pratique :



```
Developer Tools - data:text/html,chromewebdata
Elements Network Sources Timeline Profiles > X
<top frame>
var a = ["produit1", "produit2", "produit3"];
undefined
a.defineGetter_
constructor
every
filter
forEach
hasOwnProperty
indexOf
isPrototypeOf
join
lastIndexOf
length
map
pop
propertyIsEnumerable
push
reduce
reduceRight
reverse
shift
```

- une fonction permettant d'émuler un grand nombre de terminaux mobiles :



8 BabelJS et support d'ES6 côté navigateur

On a vu dans le chapitre relatif à l'héritage de classe qu'il était possible de convertir du code ES6 vers ES5, afin d'assurer une compatibilité « cross-browser » maximale.

Car si côté serveur, et notamment côté NodeJS, la norme ES6 est bien supportée, côté navigateur la situation est plus contrastée.

Mais peut-on néanmoins utiliser ES6 dans les navigateurs récents, à des fins de prototypage par exemple ? Eh bien oui, au moins pour certains navigateurs.

On notera que, à partir de 2016, il était possible de forcer certains navigateurs (notamment Firefox) à passer en mode ES6. Pour ce faire, on pouvait stipuler le code suivant dans la page HTML :

```
<script type="application/javascript;version=1.7" src="monscriptES6.js"></script>
```

Si cette solution fonctionne encore avec Firefox en ce début d'année 2017, en revanche Chrome la rejette, arguant qu'il ne supporte plus les attributs « type » exotiques au sein de la balise « script ».

Le projet BabelJS peut être utilisé côté navigateur pour convertir à la volée du code ES6 en ES5, comme dans l'exemple suivant (on notera le type « text/babel » utilisé sur le code à convertir) :

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.18.1/babel.min.js">
</script>
<script type="text/babel" src="monscriptES6.js"></script>
```

Même si la solution fonctionne, l'exécution du code est très lente, et de surcroît la version de BabelJS côté client n'est plus supportée par son auteur. On peut continuer à utiliser cette technique, mais pour des raisons évidentes de performance, mieux vaut ne pas l'encourager, et il est préférable de procéder à une conversion du code de ES6 vers ES5 côté serveur, avant d'envoyer ce code vers les navigateurs.

Pendant une période transitoire, il était possible de forcer certains navigateurs à traiter directement du code ES6, mais pour cela on avait pour obligation de placer ce code dans une IIFE, comme dans l'exemple suivant :

```
(function(){
  'use strict';

  let invk = (v) => v;

  console.log(invk(123));
})();
```

On peut voir un exemple de ce code ici : <http://jsbin.com/zejalegopo/edit?html,js,console,output>
Mais cette obligation d'utiliser une IIFE n'en est plus une en janvier 2017, au moins pour les versions les

plus récentes de Chrome et Firefox (pour les autres navigateurs, il convient de vérifier au cas par cas).

Voici à titre d'exemple un script générant un effet de type « passage en hyper-espace », via une classe ES6 et un peu de code basé sur le framework d'animation P5.JS (fonctionne dans Chrome et Firefox sans nécessiter de conversion) :

```
/*
 * Moving through space, by Konrad Junger
 * https://www.openprocessing.org/sketch/96948
 */
"use strict";

class star {
  constructor() {
    this.x = random(width);
    this.y = random(height);
    this.speed = random(0.2, 5);
    this.wachsen = parseInt(random(0, 2));
    if (this.wachsen == 1) {
      this.d = 0;
    } else {
      this.d = random(0.2, 3);
    }
    this.age = 0;
    this.sizeIncr = random(0,0.03);
  }
  render() {
    this.age++;
    if (this.age < 200){
      if (this.wachsen == 1){
        this.d += this.sizeIncr;
        if (this.d > 3 || this.d < -3) {
          this.d = 3;
        }
      } else {
        if (this.d > 3 || this.d < -3) {
          this.d = 3;
        }
        this.d = this.d + 0.2 - 0.6 * noise(this.x, this.y, frameCount);
      }
    } else {
      if (this.d > 3 || this.d < -3) {
        this.d = 3;
      }
    }
    ellipse(this.x, this.y,
      this.d*(map(noise(this.x, this.y, 0.001 * frameCount),0,1,0.2,1.5)),
      this.d*(map(noise(this.x, this.y, 0.001 * frameCount),0,1,0.2,1.5))
    );
  }
  move() {
    this.x = this.x - map(mouseX, 0, width, -0.05 * this.speed, 0.05 * this.speed) *
  }
}
```

```

        (w2 - this.x);
    this.y = this.y - map(mouseY, 0, height, -0.05 * this.speed, 0.05 * this.speed) *
        (h2 - this.y);
}
}

var neuerStern; // traduction : nouvelle étoile
var starArray = [];
var h2;//=height/2
var w2;//=width/2
var d2;//=diagonal/2
var numberOfStars = 20000;
var newStars = 50;

function setup() {
    createCanvas(900, 900);
    w2 = width / 2;
    h2 = height / 2;
    d2 = dist(0, 0, w2, h2);
    noStroke();
    neuerStern = new star();
    //frameRate(9000);
    background(0);
}
function draw() {
    fill(0, map(dist(mouseX, mouseY, w2, h2), 0, d2, 255, -10));
    rect(0, 0, width, height);
    fill(255);
    neuerStern.render();
    for (let i = 0; i < newStars; i++) { // star init
        starArray.push(new star());
    }
    for (let i = 0; i < starArray.length; i++) {
        if (starArray[i].x < 0 || starArray[i].x > width || starArray[i].y < 0
            || starArray[i].y > height) {
            starArray.splice(i, 1);
        }
        starArray[i].move();
        starArray[i].render();
    }
    if (starArray.length > numberOfStars) //{
        for (let i = 0; i < newStars; i++) {
            starArray.splice(i, 1);
        }
    }
}
}

```

Voici à titre d'exemple, le code de la classe « start » tel que BabelJS le convertit en ES5 :

```
var star = function () {
    function star() {
```

```

    _classCallCheck(this, star);

    this.x = random(width);
    this.y = random(height);
    this.speed = random(0.2, 5);
    this.wachsen = parseInt(random(0, 2)); // wachsen = grandir
    if (this.wachsen == 1) {
        this.d = 0;
    } else {
        this.d = random(0.2, 3);
    }
    this.age = 0;
    this.sizeIncr = random(0, 0.03);
}

_createClass(star, [
    key: "render",
    value: function render() {
        this.age++;
        if (this.age < 200) {
            if (this.wachsen == 1) {
                this.d += this.sizeIncr;
                if (this.d > 3 || this.d < -3) {
                    this.d = 3;
                }
            } else {
                if (this.d > 3 || this.d < -3) {
                    this.d = 3;
                }
                this.d = this.d + 0.2 - 0.6 *
                    noise(this.x, this.y, frameCount);
            }
        } else {
            if (this.d > 3 || this.d < -3) {
                this.d = 3;
            }
        }
    }

    ellipse(this.x, this.y,
        this.d * map(noise(this.x, this.y, 0.001 * frameCount),
            0, 1, 0.2, 1.5),
        this.d * map(noise(this.x, this.y, 0.001 * frameCount),
            0, 1, 0.2, 1.5)
    );
},
{
    key: "move",
    value: function move() {
        this.x = this.x - map(mouseX, 0, width, -0.05 * this.speed, 0.05 *
            this.speed) * (w2 - this.x);
        this.y = this.y - map(mouseY, 0, height, -0.05 * this.speed, 0.05 *
            this.speed) * (h2 - this.y);
    }
},
{
    key: "render",
    value: function render() {
        this.age++;
        if (this.wachsen == 1) {
            this.d += this.sizeIncr;
            if (this.d > 3 || this.d < -3) {
                this.d = 3;
            }
        } else {
            if (this.d > 3 || this.d < -3) {
                this.d = 3;
            }
            this.d = this.d + 0.2 - 0.6 *
                noise(this.x, this.y, frameCount);
        }
    }

    ellipse(this.x, this.y,
        this.d * map(noise(this.x, this.y, 0.001 * frameCount),
            0, 1, 0.2, 1.5),
        this.d * map(noise(this.x, this.y, 0.001 * frameCount),
            0, 1, 0.2, 1.5)
    );
},
{
    key: "move",
    value: function move() {
        this.x = this.x - map(mouseX, 0, width, -0.05 * this.speed, 0.05 *
            this.speed) * (w2 - this.x);
        this.y = this.y - map(mouseY, 0, height, -0.05 * this.speed, 0.05 *
            this.speed) * (h2 - this.y);
    }
}
];

```

```
        }
    ]]);
    return star;
}();
```

Pour conclure sur ce sujet :

<https://kangax.github.io/compat-table/es6/>

9 ANNEXE

9.1 Sélecteurs CSS

Tableau de synthèse des sélecteurs CSS.

Ces sélecteurs peuvent être utilisés aussi bien à l'intérieur de feuilles de styles CSS, qu'au travers des API `QuerySelector()` et `QuerySelectorAll()`.

Pattern	Ce que le pattern sélectionne...	
*	tout élément	2
E	tout élément de type E	1
E[foo]	tout élément de type E ayant un attribut "foo"	2
E[foo="bar"]	tout élément de type E ayant un attribut "foo" dont la valeur est égale à "bar"	2
E[foo~="bar"]	tout élément de type E ayant un attribut "foo" contenant une liste de valeurs séparées par des espaces, et dont l'une des valeurs est exactement égale à "bar"	2
E[foo^="bar"]	tout élément de type E ayant un attribut "foo" qui commence exactement avec la chaîne "bar"	3
E[foo\$="bar"]	tout élément de type E ayant un attribut "foo" se termine exactement à la chaîne "bar"	3
E[foo*= "bar"]	tout élément de type E ayant un attribut "foo" qui contient la chaîne "bar"	3
E[foo = "fr"]	tout élément de type E possédant un attribut "foo" qui lui-même contient une liste de valeurs séparées par des traits d'union et commençant par "fr"	2
E:root	tout élément de type E, situé à la racine du document	3
E:nth-child(n)	tout élément de type E, qui est le n-ième enfant de son parent	3
E:nth-last-child(n)	tout élément de type E, qui est le n-ième enfant de son parent en comptant à partir de la fin	3
E:nth-of-type(n)	tout élément de type E, qui est le n-ième frère de ce type	3
E:nth-last-of-type(n)	tout élément de type E, n-ième frère de son type, en partant du dernier	3
E:first-child	tout élément de type E, premier enfant de son parent	2
E:last-child	tout élément de type E, dernier enfant de son parent	3
E:first-of-type	tout élément de type E, premier frère de son type	3
E:last-of-type	tout élément de type E, dernier frère de son type	3

Pattern	Ce que le pattern sélectionne...	
*	tout élément	2
E	tout élément de type E	1
E[foo]	tout élément de type E ayant un attribut "foo"	2
E[foo="bar"]	tout élément de type E ayant un attribut "foo" dont la valeur est égale à "bar"	2
E[foo~="bar"]	tout élément de type E ayant un attribut "foo" contenant une liste de valeurs séparées par des espaces, et dont l'une des valeurs est exactement égale à "bar"	2
E[foo^="bar"]	tout élément de type E ayant un attribut "foo" qui commence exactement avec la chaîne "bar"	3
E[foo\$="bar"]	tout élément de type E ayant un attribut "foo" se termine exactement à la chaîne "bar"	3
E[foo*= "bar"]	tout élément de type E ayant un attribut "foo" qui contient la chaîne "bar"	3
E[foo = "fr"]	tout élément de type E possédant un attribut "foo" qui lui-même contient une liste de valeurs séparées par des traits d'union et commençant par "fr"	2
E:root	tout élément de type E, situé à la racine du document	3
E:nth-child(n)	tout élément de type E, qui est le n-ième enfant de son parent	3
E:nth-last-child(n)	tout élément de type E, qui est le n-ième enfant de son parent en comptant à partir de la fin	3
E:nth-of-type(n)	tout élément de type E, qui est le n-ième frère de ce type	3
E:nth-last-of-type(n)	tout élément de type E, n-ième frère de son type, en partant du dernier	3
E:first-child	tout élément de type E, premier enfant de son parent	2
E:last-child	tout élément de type E, dernier enfant de son parent	3
E:first-of-type	tout élément de type E, premier frère de son type	3
E:last-of-type	tout élément de type E, dernier frère de son type	3

E:only-child	tout élément de type E, seul enfant de son parent	3
E:only-of-type	tout élément de type E, seul frère de son type	3
E:empty	tout élément de type E qui n'a pas d'enfants (y compris qui n'a pas de noeuds de type texte)	3
E:link		
E:visited	tout élément de type E qui est l'ancre source d'un lien hypertexte qui n'est pas encore visité (: link) ou déjà visité (: visited)	1
E:active		
E:hover		
E:focus	tout élément de type E impacté par les actions en cours de l'utilisateur	1 and 2
E:target	tout élément de type E qui est la cible d'une URI de renvoi	3
E:lang(fr)	tout élément de type E dans la langue "fr" (en fonction de la langue spécifiée dans l'entête du document)	2
E:enabled		
E:disabled	tout élément d'interface utilisateur E qui est activé ou désactivé	3
E:checked	tout élément d'interface utilisateur E qui est vérifié (exemple : bouton-radio ou case à cocher) (*)	3
E::first-line	la première ligne de mise en forme d'un élément de E	1
E::first-letter	la première lettre de mise en forme d'un élément de E	1
E::before	contenu généré avant tout élément de type E	2
E::after	contenu généré après tout élément de type E	2
E.warning	tout élément de type E dont la classe est "warning"	1
E#myid	tout élément de type E avec un ID égal à "myid".	1
E:not(s)	tout élément de type E qui ne correspond au résultat renvoyé par le sélecteur s	3
E F	tout élément de type F descendant de tout élément de type E	1
E > F	un élément de type F enfant d'un élément de type E	2
E + F	tout élément de type F immédiatement précédé par tout élément de type E	2
E ~ F	tout élément de type F précédé par tout élément de type E	3

(*) Attention : on peut utiliser cette technique aussi sur les options de champs de type select (en selection multiple, c'est à dire avec l'attribut "multiple"). Exemple : dans le cas d'un champ de type "select", on peut identifier les seules options "sélectionnées" de la façon suivante :

```
var tmp = document.querySelector('select') ;
var tmp2 = tmp.querySelectorAll('option:checked'); console.dir(tmp2);
```

Les 2 syntaxes ci-dessous ne fonctionnent pas :

```
var tmp2 = tmp.querySelectorAll('option:selected'); // renvoie nodeList vide
var tmp2 = tmp.querySelectorAll('option:[selected="true"]'); // renvoie nodeList vide
```

9.2 Exercices sur les sélecteurs CSS

Le présent chapitre contient une série d'exercices destinées à renforcer la maîtrise des sélecteurs CSS.

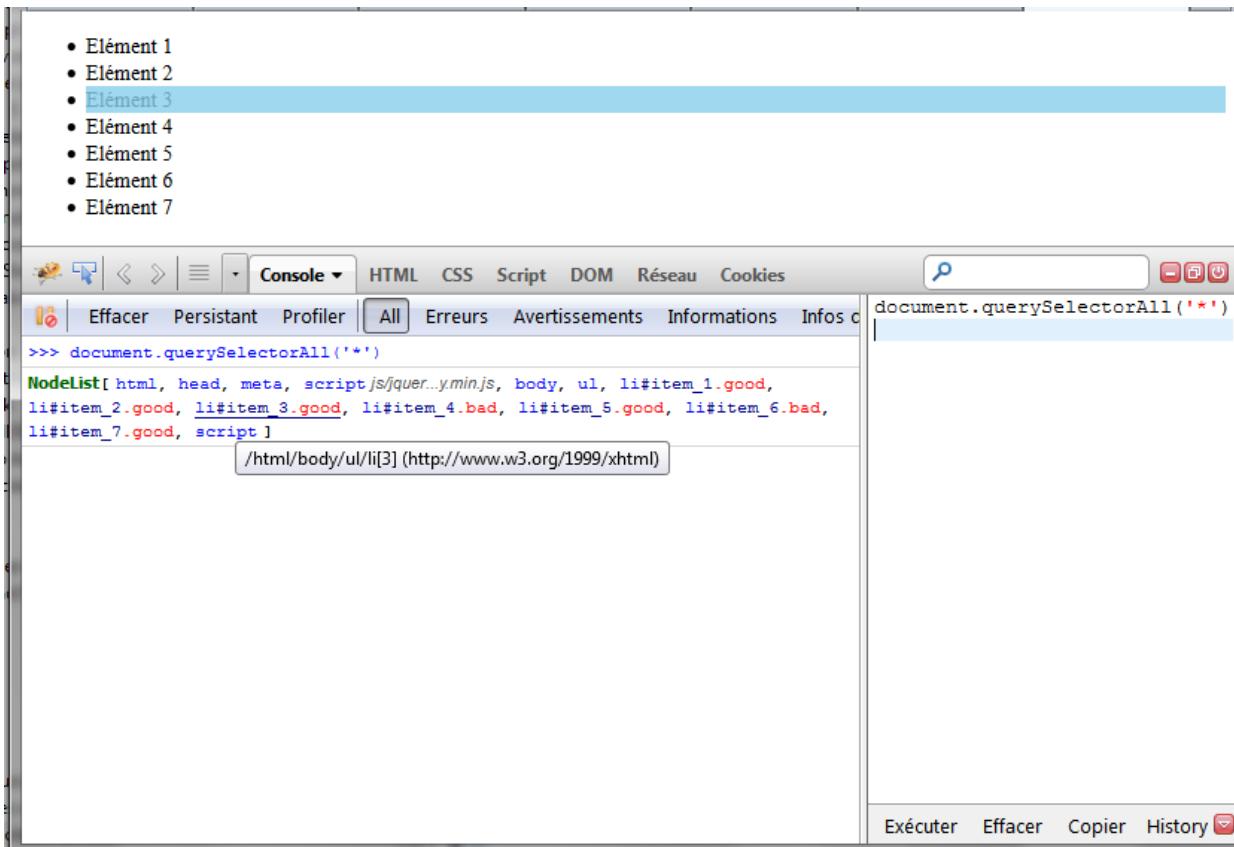
Pour chaque exercice, des équivalences seront indiquées entre le framework jQuery et les APIs `querySelector()` et `querySelectorAll()`.

On pourra effectuer les exercices en utilisant la console intégrée de Firefox :

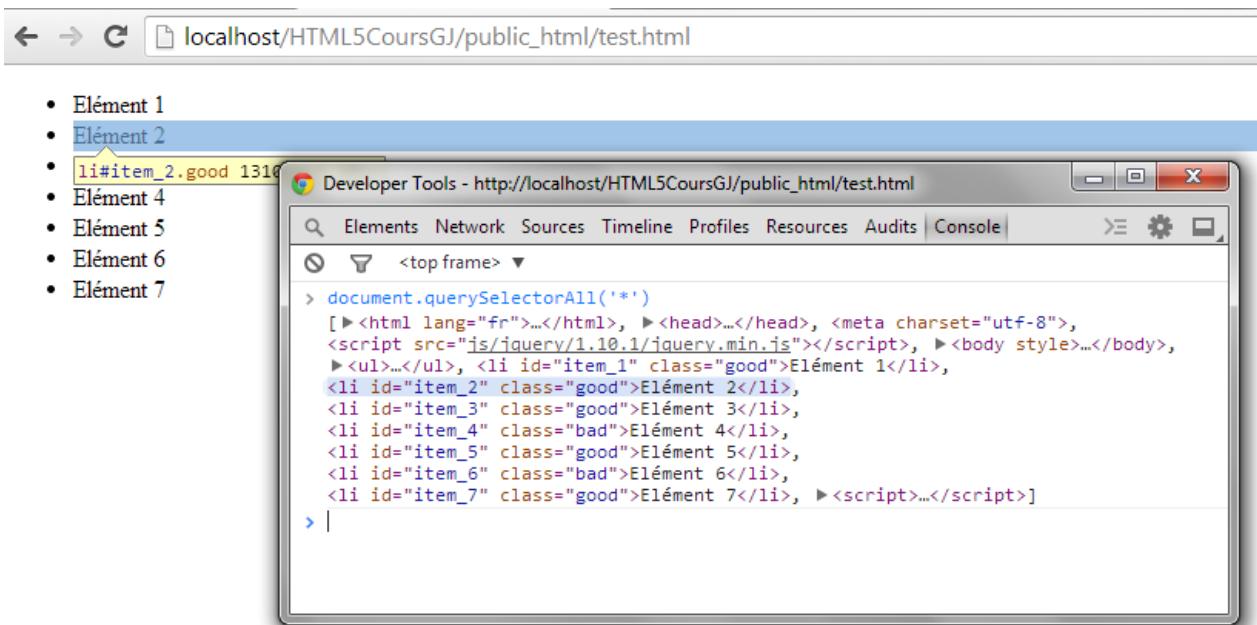
The screenshot shows the Firefox Developer Tools console interface. At the top, there is a list of seven items, each preceded by a bullet point and labeled "Elément 1" through "Elément 7". Below this is a toolbar with various icons for network, CSS, JS, security, and journal. The "CSS" icon is highlighted. The main console area displays the following log entries:

```
14:24:29,867 ◀ document.querySelector('item_1')
14:24:29,871 ▶ null
14:24:36,616 ◀ document.querySelector('#item_1')
14:24:36,619 ▶ [object HTMLLIElement]
14:24:55,425 ◀ document.querySelector('*')
14:24:55,428 ▶ [object HTMLHtmlElement]
14:25:02,910 ◀ document.querySelectorAll('*')
14:25:02,915 ▶ [object NodeList]
```

On pourra préférer utiliser le plugin Firebug :



Ou encore la console intégrée de Google Chrome :



A noter que Firebug n'offre pas dans sa version actuelle (la 1.12.6) de mécanisme d'auto-complétion sur les ordres du langage Javascript. En revanche, la console intégrée de Firefox, ainsi que celle de Google Chrome, offrent cette fonctionnalité. Par contre Firebug offre en standard une saisie de commande "multi-lignes".

9.2.1 Sélecteurs de base

Pour cette première série d'exercices sur les sélecteurs de base, nous utiliserons la page suivante :

```
<!DOCTYPE html>
<html lang="fr">
<meta charset="utf-8">
<script src="js/jquery/10.1/jquery.min.js"></script>
<body>

<ul>
  <li id="item_1" class="good">Elément 1</li>
  <li id="item_2" class="good">Elément 2</li>
  <li id="item_3" class="good">Elément 3</li>
  <li id="item_4" class="bad">Elément 4</li>
  <li id="item_5" class="good">Elément 5</li>
  <li id="item_6" class="bad">Elément 6</li>
  <li id="item_7" class="good">Elément 7</li>
</ul>

</body>
</html>
```

Note : jQuery a été intégré dans la page pour faciliter la comparaison entre jQuery et les APIs querySelector() et querySeletorAll().

Commençons par sélectionner l'élément ayant pour identifiant "item_1" :

The screenshot shows a browser's developer tools console. At the top, there is a list of four items: "Elément 1", "Elément 2", "Elément 3", and "Elément 4". Below this, the console tab is selected. The command `console.log(jQuery('#item_1'))` is entered, followed by `console.log(document.querySelector('#item_1'))` and `console.log(document.querySelectorAll('#item_1'))`. The output shows that `jQuery('#item_1')` returns an object with a single item, `document.querySelector('#item_1')` returns a single element node, and `document.querySelectorAll('#item_1')` returns a NodeList containing all four elements.

```

• Elément 1
• Elément 2
• Elément 3
• Elément 4
Elément 1
Elément 2
Elément 3
Elément 4
Elément 1
Elément 2
Elément 3
Elément 4

Console ▾ HTML CSS Script DOM Réseau Cookies 🔍
Effacer Persistant Profiler All Erreurs Avertissements
>>> console.log(jQuery('#item_1'));
console.log(document.querySelector('#item_1'));
console.log(document.querySelectorAll('#item_1'));

Object[ li#item_1.good ]
<li id="item_1" class="good">
NodeList[ li#item_1.good ]

```

On constate que, pour la même sélection :

- jQuery renvoie un tableau, que la sélection porte sur un ou plusieurs éléments, pour jQuery l'objet renvoyé sera toujours un tableau
- querySelector() renvoie un pointeur vers le premier élément trouvé correspondant à la sélection
- querySelectorAll() renvoie une NodeList qui se comporte peu ou prou comme un tableau (même si dans les faits elle n'hérite pas de toutes les méthodes liées à l'objet de type "tableau"). On rappelle que cette NodeList est statique et non pas "live", elle n'est donc pas rafraîchie dynamiquement en cas de modification du DOM.

Sélection de tous les éléments liés à la classe "good" :

The screenshot shows a browser's developer tools console. The command `console.log(jQuery('.good'))` is entered, followed by `console.log(document.querySelector('.good'))` and `console.log(document.querySelectorAll('.good'))`. The output shows that `jQuery('.good')` returns an object with four items, `document.querySelector('.good')` returns a single element node, and `document.querySelectorAll('.good')` returns a NodeList containing all four elements.

```

Effacer Persistant Profiler All Erreurs Avertissements
>>> console.log(jQuery('.good'));
console.log(document.querySelector('.good'));
console.log(document.querySelectorAll('.good'));

Object[ li#item_1.good, li#item_2.good, li#item_3.good,
li#item_5.good, li#item_7.good ]
<li id="item_1" class="good">
NodeList[ li#item_1.good, li#item_2.good, li#item_3.good,
li#item_5.good, li#item_7.good ]

```

Récupération du nombre d'éléments sélectionnés avec la méthode "length", qui fonctionne avec jQuery et querySelectorAll(), mais pas avec querySelector() :

The screenshot shows a browser's developer tools console. The command `console.log(jQuery('.good').length)` is entered, followed by `console.log(document.querySelector('.good').length)` and `console.log(document.querySelectorAll('.good').length)`. The output shows that `jQuery('.good').length` returns 5, while `document.querySelector('.good').length` and `document.querySelectorAll('.good').length` both return undefined.

```

Effacer Persistant Profiler All Erreurs Avertissements
>>> console.log(jQuery('.good').length);
console.log(document.querySelector('.good').length);
console.log(document.querySelectorAll('.good').length);

5
undefined
5

```

Pour gagner en souplesse dans la suite des exercices, on utilisera de préférence des variables pour stocker le résultat des sélections :

```

o| Effacer Persistant Profiler All Erreurs Avertissements
>>> var good1 = jQuery('.good');
console.log(good1...rySelectorAll('.good'));
console.log(good3);
Object[ li#item_1.good, li#item_2.good, li#item_3.good,
li#item_5.good, li#item_7.good ]
<li id="item_1" class="good">
NodeList[ li#item_1.good, li#item_2.good, li#item_3.good,
li#item_5.good, li#item_7.good ]

```

```

var good1 = jQuery('.good');
console.log(good1);
var good2 = document.querySelector('.good');
console.log(good2);
var good3 = document.querySelectorAll('.good');
console.log(good3);

```

La NodeList fonctionnant à peu près comme un tableau, on peut écrire ceci :

```

o| Effacer Persistant Profiler All Erreurs Avertissements
>>> var good1 = jQuery('.good');
console.log(good1...electorAll('.good'));
console.log(good3[0]);
<li id="item_1" class="good">
<li id="item_1" class="good">

```

```

var good1 = jQuery('.good');
console.log(good1[0]);
|
var good3 = document.querySelectorAll('.good');
console.log(good3[0]);

```

Sélection d'éléments de type différents :

```

o| Effacer Persistant Profiler All Erreurs Avertissements
>>> var good1 = $('#item_3 , .bad');
console.l...rAll('#item_3 , .bad');  console.log(good3);
Object[ li#item_3.good, li#item_4.bad, li#item_6.bad ]
<li id="item_3" class="good">
NodeList[ li#item_3.good, li#item_4.bad, li#item_6.bad ]

```

```

var good1 = $('#item_3 , .bad');
console.log(good1);

var good2 = document.querySelector('#item_3 , .bad');
console.log(good2);

var good3 = document.querySelectorAll('#item_3 , .bad');
console.log(good3);

```

Exemples de sélection sur la notion d'attribut (autre que "class").

On suppose tout d'abord que notre liste HTML s'est enrichie avec l'apparition de l'attribut "data-info" :

```

<ul>
  <li id="item_1" class="good" data-info="item-1">Elément 1</li>
  <li id="item_2" class="good" data-info="item-2">Elément 2</li>
  <li id="item_3" class="good" data-info="item-3">Elément 3</li>
  <li id="item_4" class="bad" data-info="item-4">Elément 4</li>
  <li id="item_5" class="good" data-info="item-5">Elément 5</li>
  <li id="item_6" class="bad" data-info="item-6">Elément 6</li>
  <li id="item_7" class="good" data-info="item-7">Elément 7</li>
</ul>

```

Sélection de tous les éléments possédant un attribut "data" ayant pour nom "data-info" et dont le contenu commence par "item" :

```

Effacer Persistant Profiler All Erreurs Avertissements Informations
>>> var test1 = jQuery('[data-info="item"]');
console.log(test1);
var test2 = document.querySelectorAll('[data-info="item"]')
console.log(test2);

Object[ li#item_1.good, li#item_2.good, li#item_3.good,
li#item_4.bad, li#item_5.good, li#item_6.bad, li#item_7.good ]
NodeList[ li#item_1.good, li#item_2.good, li#item_3.good,
li#item_4.bad, li#item_5.good, li#item_6.bad, li#item_7.good ]

```

Après quelques retouches dans la liste :

```

<ul>
  <li id="item_1" class="good" data-info="item-1">Elément 1</li>
  <li id="item_2" class="good" data-info="">Elément 2</li>
  <li id="item_3" class="good" >Elément 3</li>
  <li id="item_4" class="bad" data-info="item-4">Elément 4</li>
  <li id="item_5" class="good" data-info="item-5">Elément 5</li>
  <li id="item_6" class="bad" data-info="item-6">Elément 6</li>
  <li id="item_7" class="good" data-info="item-7">Elément 7</li>
</ul>

```

Sélection de tous les éléments qui ne possèdent pas d'attribut "data-info" :

```

Effacer Persistant Profiler All Erreurs Avertissements Informations
>>> var test1 = jQuery(':not([data-info])');
console.log(test1);
var test2 = document.querySelectorAll(':not([data-info])')
console.log(test2);

Object[ html, head, meta, script js/jquer...y.min.js, style,
body, ul, li#item_3.good, script ]
NodeList[ html, head, meta, script js/jquer...y.min.js, style, body,
ul, li#item_3.good, script ]

```

Sélection - uniquement à l'intérieur de la liste "ul > li" - de tous les éléments qui ne possèdent pas d'attribut "data-info" :

```

Effacer Persistant Profiler All Erreurs Avertissements Informations
>>> var test1 = jQuery('li:not([data-info])');
console.log(test1);
var test2 = document.querySelectorAll('li:not([data-info])')
console.log(test2);

Object[ li#item_3.good ]
NodeList[ li#item_3.good ]

```

Suggestion : en vous appuyant sur le tableau des sélecteurs CSS fourni en annexe, amusez-vous à tester d'autres critères de recherche.

9.2.2 Sélecteurs descendants

Exercices pour apprendre à se promener dans l'arborescence du DOM.

Soit la liste imbriquée suivante :

```
<ul>
  <li class="lvl1">Elément 1</li>
  <li class="lvl1" id="item2">Elément 2
    <ul>
      <li class="lvl2">Sous-élément 1</li>
      <li class="lvl2">Sous-élément 2
        <ul>
          <li class="lvl3">Sous-élément 2-1</li>
        </ul>
      </li>
      <li class="lvl2">Sous-élément 3</li>
    </ul>
  </li>
  <li class="lvl1">Elément 3</li>
  <li class="lvl1">Elément 4</li>
</ul>
```

Commençons par sélectionner tous les "li" dépendant d'un "li" de niveau supérieur :

```
var test1 = jQuery('li li');
console.log(test1); // Object[li lvl2, li lvl2, li lvl3, li lvl2]
var test2 = document.querySelectorAll('li li') ;
console.log(test2); // NodeList[li lvl2, li lvl2, li lvl3, li lvl2]
```

La sélection ci-dessous en revanche ne renvoie rien :

```
var test1 = jQuery('li > li');
console.log(test1); // Object[]
var test2 = document.querySelectorAll('li > li') ;
console.log(test2); // NodeList[]
```

Par contre, celle-ci fonctionne correctement :

```
var test1 = jQuery('li > ul > li');
console.log(test1); // Object[li lvl2, li lvl2, li lvl3, li lvl2]
var test2 = document.querySelectorAll('li > ul > li') ;
console.log(test2); // NodeList[li lvl2, li lvl2, li lvl3, li lvl2]
```

Si on souhaite démarrer notre sélection à partir d'un élément "li" appartenant à la classe "lvl2", on écrira ceci :

```
var test1 = jQuery('li.lvl1 > ul > li');
console.log(test1); // Object[li.lvl2, li.lvl2, li.lvl2]
var test2 = document.querySelectorAll('li.lvl1 > ul > li') ;
console.log(test2); // NodeList[li.lvl2, li.lvl2, li.lvl2]
```

ATTENTION : dans la suite des exercices, on ne donnera plus que rarement les équivalences avec jQuery, puisque vous avez compris que les résultats obtenus à jQuery sont en règle générale équivalents à ceux obtenus avec querySelectorAll().

Si on souhaite sélectionner l'élément "li" de même niveau que l'élément "li" ayant pour identifiant "item2", on pourra écrire (2 possibilités) :

```
var test = document.querySelectorAll('#item2 + li') ;
console.log(test[0].innerHTML); // Elément 3

var test = document.querySelectorAll('#item2 + li') ;
console.log(test[0].innerHTML); // Elément 3
```

IMPORTANT : certaines techniques de sélection propres à jQuery ne sont pas conformes à la norme CSS3, et ne seront pas acceptées par la querySelectorAll(). Dans l'exemple ci-dessous, le critère de sélection '.lvl1:eq(1)' utilisé avec jQuery n'est pas conforme à la norme CSS3 :

```
var test = document.querySelectorAll('.lvl1:nth-child(2)');
console.log(test); // NodeList[li#item2.lvl1]

var test2 = jQuery('.lvl1:eq(1)'); // équivalent à jQuery('.lvl1:nth-child(2)');
console.log(test2); // Object[li#item2.lvl1]
```

Autre exemple de syntaxe jQuery non conforme à CSS3 :

```
var test = document.querySelectorAll('.lvl1:first-of-type');
console.log(test); // NodeList[li.lvl1]

var test2 = jQuery('.lvl1:first'); // équivalent à jQuery('.first-of-type') ;
console.log(test2); // Object[li.lvl1]
```

jQuery dans ses versions les plus récentes (au moins à partir de la 1.10, et probablement avant), reconnaît parfaitement les sélecteurs CSS3, il est donc recommandé de s'en tenir à la norme CSS3 et d'éviter autant que possible d'utiliser certaines techniques spécifiques à jQuery.

Technique de sélection plus sophistiquée qui fonctionne aussi bien avec querySelectorAll() et

jQuery : on souhaite ici sélectionner le dernier élément de la sous-liste de classe "lvl2" :

```
var test = document.querySelectorAll('.lvl1:nth-child(2) ul .lvl2:last-child');
console.log(test[0].innerHTML); // Sous-élément 3
```

Prenez un moment pour analyser et bien comprendre le cas ci-dessus.

Autre exemple à tester et analyser :

```
var test = document.querySelectorAll('.lvl1:nth-child(2) ul .lvl2:nth-child(3)');
console.log(test[0].innerHTML); // Sous-élément 3
```

Autre exemple :

```
var test = document.querySelectorAll('.lvl1:nth-child(2) ul .lvl2:nth-child(2) ul
li:first-child');
console.log(test[0].innerHTML); // Sous-élément 2-1
```

Autre exemple de sélecteur jQuery non conforme CSS3, mais avec une solution de contournement permettant d'aboutir au même résultat avec querySelector() :

```
var test2 = jQuery('li.lvl2:parent');
console.log(test2); // Object[li.lvl2, li.lvl2, li.lvl2]

var test = document.querySelector('li.lvl2');
console.log(test.parentNode.children); // HTMLCollection[li.lvl2, li.lvl2,
li.lvl2]
```

9.2.3 Sélecteurs sur formulaires

Soit le formulaire suivant :

```
<style>
  div {
    float: left;
    width: 49%;
  }
</style>
<body>

  <form id="myForm">
    <div>
      <label for="firstName">Nom</label>
      <input type="text" id="firstName" name="firstName" required><br />
      <label for="LastName">Prénom</label>
      <input type="text" id="LastName" name="LastName"><br />
      <label for="myPassword">Mot de passe</label>
      <input type="password" id="myPassword" name="myPassword"
disabled="disabled"><br />
      <label for="imgName">Fichier image</label><br />
      <input type="file" id="imgName" name="imgName"><br />
      <button id="myButton">
        buttons <span style="color: red">ALLOW</span> HTML
      </button>
      <br />
    </div>
    <div>
      <fieldset>
        <legend>Boutons Radios</legend>
        <label for="rb_1">Oui</label>
        <input type="radio" id="rb_1" name="rb" checked="checked"><br />
        <label for="rb_2">Non</label>
        <input type="radio" id="rb_2" name="rb"><br />
      </fieldset>
      <fieldset>
        <legend>Cases à cocher</legend>
        <label for="cb_1">Wrapped</label>
        <input type="checkbox" id="cb_1" name="cb" checked="checked"><br />
      </fieldset>
      <label for="cb_2">Card</label>
      <input type="checkbox" id="cb_2" name="cb"><br />
    </fieldset>
    <label for="mySelection">Liste de sélection</label><br />
    <select id="mySelection" name="mySelection">
      <option>Rouge</option>
      <option selected="selected">Vert</option>
      <option>Bleu</option>
    </select>
  </form>
</body>
```

```

        </select><br /> Email (HTML5 Inputs)<br />
        <input type="email" id="myEmail" name="myEmail"><br />
    </div>
    <div style="clear: both;"></div>
    <input type="reset" id="myReset">
    <input type="submit" id="mySubmit"><br />
</form>

```

Commençons par sélectionner tous les éléments de formulaire de type "text" :

```

var test = document.querySelectorAll('input[type="text"]');
console.log(test); /* NodeList[input#firstName property value = "" attribute value =
"null", input#lastName property value = "" attribute value = "null"] */

```

Si on a plusieurs formulaires sur la page, on peut préciser l'identifiant du formulaire dans le critère de sélection :

```

var test = document.querySelectorAll('#myForm input[type="text"]');
console.log(test); /* NodeList[input#firstName property value = "" attribute value =
"null", input#lastName property value = "" attribute value = "null"] */

```

Un seul champ est obligatoire dans ce formulaire (cf. l'attribut HTML5 "required" sur le champ "firstName"). On peut le sélectionner en ajoutant la pseudo-classe CSS ":required" dans la chaîne de recherche :

```

var test = document.querySelectorAll('#myForm input[type="text"]:required');
console.log(test); // NodeList[input#firstName property value = "" attribute value =
"null"]

```

On peut aussi utiliser les pseudo-classes CSS ":enabled" et ":disabled" pour sélectionner les champs de saisie actif ou inactifs :

```

var test = document.querySelectorAll('#myForm input[type="text"]:disabled');
console.log(test); // NodeList[]

```

Tiens, ça ne marche pas ici, pouvez-vous dire pourquoi ?

Vous pouvez aussi utiliser le mot clé ":checked" pour sélectionner les champs de type "bouton radio" et "case à cocher" qui sont cochés dans le formulaire.

Vous pouvez aussi utiliser le mot clé ":selected" pour sélectionner les champs de type "liste déroulante" qui ont une valeur sélectionnée.

IMPORTANT : il existe quelques différences entre jQuery et querySelectorAll(). Par exemple la technique de sélection ci-dessous fonctionne avec jQuery mais pas avec querySelectorAll() :

```
var test = document.querySelectorAll('#myForm :input'); // ne fonctionne pas
(sélecteur non conforme CSS3)
console.log(test); // ne renverra rien car la ligne précédente "plante"
l'interpréteur Javascript

var test2 = jQuery('#myForm :input'); // fonctionne très bien avec jQuery
console.log(test2); // renvoie un objet de type tableau contenant tous les champs du
formulaire
```

jQuery autorise l'utilisation de filtres tels que ":input", ":radio", ou encore ":checkbox", mais ces filtres ne sont pas conformes à la spécification de la norme CSS3.

9.3 La librairie Underscore.JS

Site officiel de la librairie Underscore.JS :

<http://underscorejs.org/>

Cette librairie Javascript propose près d'une centaine de fonctions prenant en charge un certain nombre de problématiques auxquelles sont confrontés les développeurs Javascript au quotidien.

On y trouve des fonctions dédiées à la manipulation de collections, de tableaux, d'objets, et des fonctions utilitaires.

Collections	Tableaux	Objets	Utilitaires
- each	- first	- bind	- noConflict
- map	- initial	- bindAll	- identity
- reduce	- last	- partial	- constant
- reduceRight	- rest	- memoize	- noop
- find	- compact	- delay	- times
- filter	- flatten	- defer	- random
- where	- without	- throttle	- mixin
- findWhere	- union	- debounce	- iteratee
- reject	- intersection	- once	- uniqueId
- every	- difference	- after	- escape
- some	- uniq	- before	- unescape
- contains	- zip	- wrap	- result
- invoke	- unzip	- negate	- now
- pluck	- object	- compose	- template
- max	- indexOf		
- min	- lastIndexOf		
- sortBy	- sortedIndex		
- groupBy	- findIndex		
- indexBy	- findLastIndex		
- countBy	- range		
- shuffle			
- sample			
- toArray			
- size			
- partition			

TODO : compléter ce chapitre avec quelques exemples d'utilisation

9.4 Quelques outils à connaître (*jslint* et *jshint*)

Deux outils en ligne existent pour aider le développeur Javascript à écrire un code de qualité : JSLint et JSHint.

<http://www.jslint.com>

<http://www.jshint.com>

JSLint est développé par Douglas Crockford, un expert Javascript mondialement connu, auteur du best-seller "Javascript, the good parts".

JSLint peut s'utiliser en ligne et offre de nombreuses options pour activer ou désactiver certaines alertes. JSLint est écrit en Javascript, il s'appuie sur un certain nombre de recommandations définies par Douglas Crockford lui-même, recommandations que l'on peut consulter ici :

<http://javascript.crockford.com/code.html>

JSHint est un fork de JSLint, développé par Anton Kovalyov. Le paramétrage par défaut des alertes dans JSHint diffère quelque peu de celui de JSLint, mais surtout il est beaucoup plus paramétrable. De plus, JSHint offre une interface plus conviviale, et la possibilité de le télécharger sous la forme d'un plugin (plusieurs versions sont prévues pour différents éditeurs de code, dont SublimeText et Visual Studio).

9.5 Extraire les paramètres d'une URL

L'extraction des paramètres d'une URL n'est pas l'opération la plus simple qui soit en JS. On peut s'en rendre compte en lisant le grand nombre de solutions proposées ici :

<http://stackoverflow.com/questions/901115/how-can-i-get-query-string-values-in-javascript>

La première solution proposée peut répondre à la plupart des besoins :

```
function getParameterByName(name, url) {
    if (!url) {
        url = window.location.href;
    }
    name = name.replace(/[\[\]]/g, "\\$&");
    var regex = new RegExp("[?&]" + name + "=([^\#]*|(&|#|$))",
        results = regex.exec(url);
    if (!results) return null;
    if (!results[2]) return '';
    return decodeURIComponent(results[2].replace(/\+/g, " "));
}

console.log(getParameterByName('titi'));
```

Pour des solutions plus avancées, on aura certainement intérêt à se tourner vers l'un des projets suivants (non testés) :

<http://medialize.github.io/URI.js/>

<https://github.com/stretchr/arg.js>

9.6 Tracé d'un épicycloïde avec Canvas

Exemple de code HTML et JS permettant de tracer des formes graphiques de type épicycloïde

Le principe est ici de pouvoir tracer un graphe soit en mode instantané, soit en mode progressif. Dans les 2 cas, les valeurs ayant servi à calculer le graphe sont conservées sur le canvas au moyen d'attributs data-* (un attribut par champ de saisie).

Si l'utilisateur souhaite retrouver les valeurs ayant servi à générer un graphe, il lui suffit de cliquer sur le graphe pour « recharger » les champs de saisie à partir des attributs data-* correspondants.

Voici le formulaire de création d'un épicycloïde (stylé avec Bootstrap) :

Graphique Epicycloïde

Rayon de la forme (en pixels)

Nombre de points :

Nombre de rebroussements :

Couleur du tracé :

Couleur du fond :

Mode de tracé :

Tracer

Code source du formulaire :

```

<!-- Begin page content -->
<div class="container">
  <br><br>
  <div class="page-header">
    <h2>Graphique Epicycloïde</h2>
  </div>
  <div>
    <table>
      <tbody>
        <tr>
          <td><label for="rayon_forme" class="lead">Rayon de la forme (en pixels)</label></td>
          <td><select name="rayon_forme" id="rayon_forme"
                     required="required" class="form-control">
            <option value="100">100</option>
            <option value="150">150</option>
            <option value="200" selected="selected">200</option>
            <option value="300">300</option>
            <option value="400">400</option>
            <option value="500">500</option>
            <option value="600">600</option>
            <option value="700">700</option>
            <option value="800">800</option>
          </select>
        </td>
      </tr>
      <tr>
        <td><label for="nbriter" class="lead">Nombre de points :</label></td>
        <td><input id="nbriter" name="nbriter" min="1"
                   max="800" step="1" value="200" type="number" class="form-control"></td>
      </tr>
      <tr>
        <td><label for="nbrebro" class="lead">Nombre de rebroussements :</label></td>
        <td><input id="nbrebro" name="nbrebro" min="1"
                   max="800" step="1" value="2" type="number" class="form-control"></td>
      </tr>
      <tr>
        <td><label for="couleur" class="lead">Couleur du tracé :</label></td>
        <td><input id="couleur" name="couleur"
                   value="#eb1212" type="color" class="form-control"> </td>
      </tr>
      <tr>
        <td><label for="couleur_fond" class="lead">Couleur du fond :</label></td>
        <td><input id="couleur_fond" name="couleur"
                   value="#FFFFFF" type="color" class="form-control"> </td>
      </tr>
      <tr>
        <td><label for="mode_trace" class="lead">Mode de tracé :</label></td>
        <td><select name="mode_trace" id="mode_trace" required="required" class="form-control">
          <option value="1">Tracé instantané</option>
          <option value="2">Tracé progressif</option>
        </select></td>
      </tr>
    </tbody>
  </table>
  <br> <br>
  <button type="button" id="submitbutton" class="btn btn-lg btn-primary">Tracer</button>
</div>
</div>

<div id="image">
</div>

```

Code JS :

```

/**
 * Fonction drawProg
 * Fonction dédiée au dessin de graphes en mode progressif
 * Nécessite pour fonctionner que l'objet MathGraph soit dans
 * un "scope" facile d'accès (en l'occurrence le scope "window"),
 * sinon l'utilisation de la fonction requestAnimationFrame()
 * se transforme en usine à gaz.
 * @returns {undefined}
 */
var drawProg = function () {
    "use strict";
    var tmp = MathGraph.coords[MathGraph.i_draw];
    MathGraph.context.lineCap = 'round';
    MathGraph.context.moveTo(tmp.x1, tmp.y1);
    MathGraph.context.lineTo(tmp.x2, tmp.y2);
    MathGraph.context.stroke();
    MathGraph.i_draw += 1;
    if (MathGraph.i_draw < MathGraph.nb_items) {
        MathGraph.idanim = window.requestAnimationFrame(drawProg, MathGraph.canvas);
    } else {
        MathGraph.context.closePath();
        window.cancelAnimationFrame(MathGraph.idanim);
        //setInterval(rotate, 100);
        //MathGraph.idanim2 = window.requestAnimationFrame(rotate, MathGraph.canvas);
    }
};

function rotate() {
    "use strict";
    var canvasWidth = 400 ;
    var canvasHeight = 400 ;
    //MathGraph.context.save();
    // Clear the canvas
    MathGraph.context.clearRect(0, 0, canvasWidth, canvasHeight);

    // Move registration point to the center of the canvas
    MathGraph.context.translate(canvasWidth/2, canvasWidth/2);

    // Rotate 1 degree
    MathGraph.context.rotate(Math.PI / 180);

    // Move registration point back to the top left corner of canvas
    MathGraph.context.translate(-canvasWidth/2, -canvasWidth/2);
    //MathGraph.context.restore();
}

/**
 * Fonction canvasGetDataset
 * Reprend le dataset défini sur un canvas et utilise les valeurs
 * définies dans ce dataset pour rafraîchir le formulaire permettant
 * de générer les graphes
 * @param {type} canvasref
 * @returns {Boolean}
 */
var canvasGetDataset = function (canvasref) {
    "use strict";
    var inputs = document.querySelectorAll('input, select');
    var dataset_items = canvasref.dataset ;
    var dataset_keys = Object.getOwnPropertyNames(dataset_items);
    var dataset_length = dataset_keys.length ;
    var i = 0 ;
    var form_values = {} ;
    for (i = 0 ; i < dataset_length ; i+=1) {
        form_values[dataset_keys[i]] = dataset_items[dataset_keys[i]];
    }
}

```

```

var inputs = document.querySelectorAll('input, select');
var xtmp = 0;
var ztmp = 0;
var xsize = inputs.length;
var seloption = null ;
var current_val = null ;
for (xtmp = 0; xtmp < xsize; xtmp += 1) {
    current_val = form_values[inputs[xtmp].id] ;
    if (inputs[xtmp].tagName === 'INPUT') {
        inputs[xtmp].value = current_val ;
    } else {
        if (inputs[xtmp].tagName === 'SELECT') {
            seloption = inputs[xtmp].options ;
            for (ztmp = 0; ztmp < seloption.length; ztmp += 1) {
                if (seloption[ztmp].value == current_val) {
                    seloption[ztmp].selected = true ;
                    break ;
                }
            }
        }
    }
}
return true ;
};

/**
 * On place un évènement sur le clic du bouton "Tracer"
 * @returns {undefined}
 */
function graph_event() {
    "use strict";
    document.querySelector('#submitbutton').addEventListener('click',
        function (event) {
            var inputs = document.querySelectorAll('input, select');
            MathGraph.init('image', inputs);
            MathGraph.calc();
        }, false
    );
}

/**
 * Création de l'objet Epicycloide
 * @returns {EpicycloideDraw}
 */
var EpicycloideDraw = function () {
    "use strict";

    /**
     * Methode d'initialisation de l'objet EpicycloideDraw
     * @param {String} div_id
     * @param {NodeList} inputs
     * @returns {Boolean}
     */
    this.init = function (div_id, inputs) {
        this.nbtriangles = null;
        this.divcote = null;
        this.nbiter = null;
        this.rotation = null;

        this.params = {};

        this.pointer = null;
        this.canvas = null;
        this.context = null;

        this.nb_items = 0;
    };
}

```

```

this.idanim = 0;
this.i_draw = 0;

this.xcentre = 0;
this.ycentre = 0;
this.xdecalage = 0;
this.ydecalage = 0;

this.width = 0;
this.height = 0;

this.coords = [];// utilisation uniquement dans le cas de tracé progressif

var xtmp = 0;
var xsize = inputs.length;

/*
 * Récupération des champs de formulaires dans le tableau "params"
 */
for (xtmp = 0; xtmp < xsize; xtmp += 1) {
    this.params[inputs[xtmp].id] = inputs[xtmp].value;
}

this.xcentre = Number(this.params['rayon_forme']);
this.ycentre = this.xcentre;
this.xdecalage = this.xcentre * 1.1;
this.ydecalage = this.xcentre;

this.width = this.xcentre * 2 + 50;
this.height = this.ycentre * 2 + 50;

this.pointer = document.getElementById(div_id);
this.canvas = document.createElement('canvas');
//this.pointer.appendChild(this.canvas);
this.pointer.insertBefore(this.canvas, this.pointer.firstChild) ;

this.canvas.width = this.width;
this.canvas.height = this.height;
this.context = this.canvas.getContext("2d");

/*
 * On place les valeurs des formulaires en attributs de type
 * "dataset" sur les canvas générés, de manière à conserver une trace
 * des valeurs ayant servi à générer chaque graphe.
 */
for (xtmp = 0; xtmp < xsize; xtmp += 1) {
    this.canvas.dataset[inputs[xtmp].id] = inputs[xtmp].value;
}

/*
 * On ajoute un évènement de type click sur le canvas, pour pouvoir
 * récupérer ses paramètres (stockés dans dataset) sur le formulaire
 * de demande de génération de graphe
 */
this.canvas.addEventListener('click',
    function (event) {
        canvasGetDataset(event.currentTarget) ;
    }, false
);

return true;
}

```

```


/**
 * Méthode dédiée au calcul du graphe
 * Cette méthode effectue également le tracé du graphe, si le
 * mode de tracé "immédiat" a été demandé (dans le cas contraire,
 * c'est la fonction drawProg() qui réalise le tracé).
 * @returns {Boolean}
 */
this.calc = function () {
    var x = [];
    var y = [];
    var rayon = Number(this.params['rayon_forme']);
    var nbiter = Number(this.params['nbiter']);
    var rebrous = Number(this.params['nbrebro']);
    var couleur = this.params['couleur'];
    var couleur_fond = this.params['couleur_fond'];
    var mode_trace = this.params['mode_trace'];
    var i, i2, j = 0;
    var ti, tj = 0;
    var x1, x2, y1, y2 = 0;

    this.context.beginPath();
    this.context.rect(0, 0, this.canvas.width, this.canvas.height);
    this.context.fillStyle = couleur_fond;
    this.context.fill();

    this.context.strokeStyle = couleur;
    this.context.lineWidth = 1;
    this.context.lineCap = 'square';
    this.context.beginPath();

    rebrous += 1;
    for (i = 1; i <= nbiter - 1; i+=1) {
        ti = 2 * Math.PI * i / nbiter;
        j = i * rebrous - nbiter * Math.floor(i * rebrous / nbiter);
        tj = 2 * Math.PI * j / nbiter;
        x1 = Math.round(this.xcentre + rayon * Math.cos(ti));
        y1 = Math.round(this.ycentre + rayon * Math.sin(ti));
        x2 = Math.round(this.xcentre + rayon * Math.cos(tj));
        y2 = Math.round(this.ycentre + rayon * Math.sin(tj));
        if (mode_trace == '1') {
            this.context.moveTo(x1, y1);
            this.context.lineTo(x2, y2);
        } else {
            this.coords.push({'x1': x1, 'y1': y1, 'x2': x2, 'y2': y2});
        }
    }

    this.nb_items = this.coords.length;
    if (this.nb_items > 0) {
        this.i_draw = 0;
        drawProg();
    }

    this.context.stroke();
    this.context.closePath();

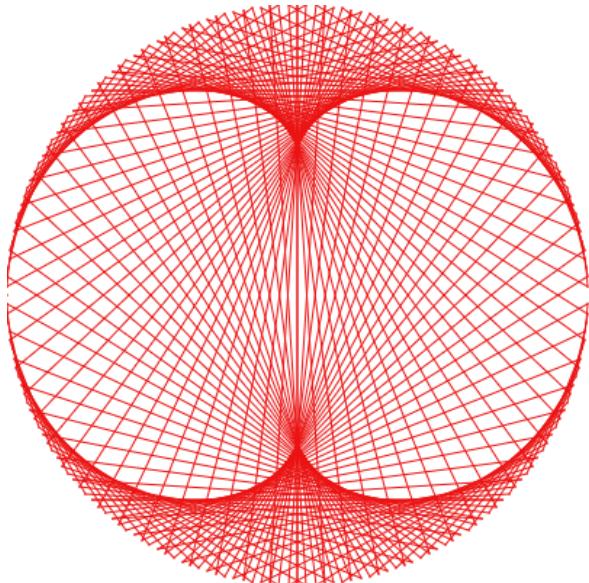
    return true;
}
}


```

```
/*
 * On est obligé de déclarer l'objet MathGraph globalement pour
 * faciliter son utilisation dans la fonction drawProg().
 * Si on ne procède pas ainsi, l'utilisation dans drawProg() de la fonction
 * requestAnimationFrame() devient complexe (problème de "scope").
 */
var MathGraph = new EpicycloideDraw();

window.onload = function() {
    graph event();
}
```

Résultat final :



9.7 Exemple de drag and drop entre listes HTML

Todo : chapitre à détailler dans une prochaine version de ce support

Exemple fonctionnel fourni en cours avec un « projet exemple ».

9.x Exemple de templating avec le projet HandleBars.JS

Todo : chapitre à détailler dans une prochaine version de ce support

Exemple fonctionnel fourni en cours avec un « projet exemple ».

9.x Tri de liste avec Tablesort

Todo : chapitre à détailler dans une prochaine version de ce support

Exemple fonctionnel fourni en cours avec un « projet exemple ».

9.x Codepen.io et jsbin, les amis du prototypeur

Todo : chapitre à détailler dans une prochaine version de ce support

9.x La librairie Moment.js

Todo : chapitre à détailler dans une prochaine version de ce support

9.x La librairie Math.js

Todo : chapitre à détailler dans une prochaine version de ce support

9.x Exemple de fenêtre modale avec Bootstrap

Todo : chapitre à détailler dans une prochaine version de ce support

10 Bibliographie

Liste (non exhaustive) d'ouvrages qui ont été utilisés pour la création de ce cours :

- "Dom Scripting", de Jeremy Keith et Jeffrey Sambells, ed. Friendsof 2010 (Apress)
- "Javascript Patterns", de Stoyan Stefanov (O'Reilly)
- "Learning the Javascript Design Patterns", de Addy Osmani (O'Reilly)
- "You don't know JS : this and Object Prototypes", de Kyle Simpson (O'Reilly)
- "You don't know JS : Scope and Closures", de Kyle Simpson (O'Reilly)
- "You don't know JS : Types and Grammar", de Kyle Simpson (O'Reilly)
- "You don't know JS : Async and Performance", de Kyle Simpson (O'Reilly)
- "The Modern Web", de Peter Gasston (NoStarch Press)
- "Learning JavaScript : A Hands-On Guide to the Fundamentals of Modern JavaScript", de Tim Wright (Addison-Wesley)
- "Solutions HTML 5", de Marco Casario...(*collectif*), (Pearson, disponible en français)
- "Dive into HTML 5", de Mark Pilgrim (livre disponible en "open-source")
- "Javascript, the good parts", de Douglas Crockford (O'Reilly)
- "Javascript Cookbook", de Shelley Powers (O'Reilly)
- "Maintainable Javascript", de Nicholas C. Zakas (O'Reilly)
- "DOM Enlightenment", de Cody Lindley (O'Reilly)

Pour un tour d'horizon complet de la nouvelle norme ES6 :

- "Exploring ES6", Axel Rauschmayer
 - o <http://exploringjs.com>
- "Understanding EcmaScript 6", Nicolas Zakas
 - o <https://leanpub.com/understandinges6/read/>

Front-end Handbook 2017 (free ebook), by Cody Lindley :

<https://www.gitbook.com/book/frontendmasters/front-end-handbook-2017/>

JavaScript Stack from Scratch :

<https://github.com/verekia/js-stack-from-scratch>

11 Liens utiles, articles

Exemples d'utilisation de la fonction each() de jQuery :

<http://www.sitepoint.com/jquery-each-function-examples/>

Comparatif entre VanillaJS et jQuery :

<http://putaindecode.io/fr/articles/js/de-jquery-a-vanillajs/>

Il faut souligner l'arrivée d'une nouvelle tendance en 2015 : VanillaJS. Il ne s'agit pas d'un nouveau framework, le terme VanillaJS désigne le fait de coder en « pur » JS, ou encore en « JS natif », bref d'utiliser au maximum les APIs du HTML5 et de se passer le plus possible des frameworks. Cette tendance est apparue notamment avec la montée en puissance de JS au sein des terminaux mobiles (smartphones, tablettes..). Ces terminaux disposant de moins de mémoire qu'un PC de bureau, et bénéficiant souvent de bande passante moins large, la nécessité de coder en pur JS et d'éviter de charger des frameworks et composants superflus, est devenue particulièrement sensible.

Comment se passer de jQuery avec VanillaJS :

<http://myclientisrich-leblog.com/2013/09/parlons-dev-se-passer-de-jquery-avec-vanilla/>

<http://clubmate.fi/jquerys-closest-function-and-pure-javascript-alternatives/>

Exemple de projets développés selon une approche VanillaJS :

- Le projet Tablesort, un « Datagrid » équivalent au projet Datatables, mais sans jQuery :
 - o <https://github.com/tristen/tablesort>
- Micro-templating écrit par John Resig (le « papa » de jQuery) :
 - o <http://ejohn.org/blog/javascript-micro-templating/>
- FlatPicker, un date picker écrit en VanillaJS
 - o <https://chmln.github.io/flatpickr/>

Très intéressant slide de Nicholas Zakas présentant différentes techniques de manipulation du DOM :

<http://fr.slideshare.net/nzakas/javascript-apis-youve-never-heard-of-and-some-you-have/>

Liens utiles autour des évolutions relatives à ES6, aux nouveaux outils Javascript, etc... :

<http://www.telerik.com/campaigns/kendo-ui/javascript-future>

<http://developer.telerik.com/featured/tools-learn-javascript/>

<https://www.nczonline.net/blog/2016/04/es6-module-loading-more-complicated-than-you-think/>

11 Changelog

Version 1.1

- Ajout du chapitre 3.9.7 (Héritage)

Version 1.2

- Compléments sur le pseudo-sélecteur « checked » dans le chapitre 9.1 (Sélecteurs CSS)
- Réécriture du chapitre 6.5 (IndexedDB et WebSQL)
- Compléments sur le chapitre 10 (Bibliographie)

Version 1.3

- Réécriture du chapitre 4.5.7 (DomContentLoaded)
- Compléments sur le chapitre 11 (Liens utiles)
- Ajout du chapitre 6.5 (Websockets, Webworkers, ..)