

Ecole Centrale de Lyon

U.E. Informatique

ELC-D3 - Applications Web

Javascript, le langage

Daniel Muller
Daniel.Muller@ec-lyon.fr



2. Javascript, le langage

Plan du cours

- 2.1 Qu'est-ce que Javascript ?
 - 2.1.1 Contextes d'utilisation
 - 2.1.2 Javascript n'est pas Java
 - 2.1.3 Historique
- 2.2 Démarrage rapide
 - 2.2.1 Outils de développement
 - 2.2.2 Les outils du navigateur
 - 2.2.3 Exercice : prise en main de la console
 - 2.2.4 HTML et la balise script
 - 2.2.5 Afficher du texte avec document.write
 - 2.2.6 Modifier l'interface avec element.innerHTML
 - 2.2.7 Afficher dans un popup avec alert
 - 2.2.8 Les méthodes de la console
 - 2.2.9 Exercice : affichage d'information
- 2.3 Caractéristiques générales du langage
 - 2.3.1 Structure du code
 - 2.3.2 Point virgule
 - 2.3.3 Définition de fonctions
 - 2.3.4 Portée des variables
 - 2.3.5 Protection de l'espace global
 - 2.3.6 Types primitifs
 - 2.3.7 Typage dynamique
 - 2.3.8 Conversions de type
 - 2.3.9 Tableaux
 - 2.3.10 Exercice : prise en main de Javascript
- 2.4 Orientation Objet
 - 2.4.1 Tableaux associatifs
 - 2.4.2 JSON
 - 2.4.3 Objets
 - 2.4.4 Objets natifs
 - 2.4.5 Typage canard
 - 2.4.6 Objet global
 - 2.4.7 Exercice : utilisation des objets prédéfinis.

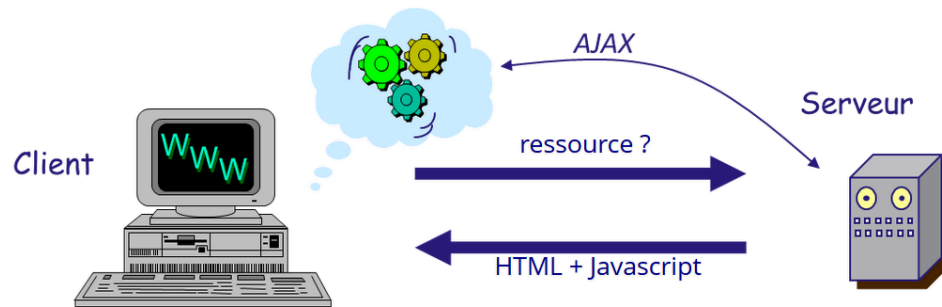


- 2.5 Fonctions
 - 2.5.1 Fonctions de première classe !
 - 2.5.2 Méthodes
 - 2.5.3 Fonctions imbriquées
 - 2.5.4 Clôtures
 - 2.5.5 Prototypes
 - 2.5.6 Constructeurs
 - 2.5.7 Classes
 - 2.5.8 Retour sur this
 - 2.5.9 Exercice : Calcul de moyennes
- 2.6 Rapide introduction au DOM
 - 2.6.1 Manipulation du DOM
 - 2.6.2 Gestion des événements
 - 2.6.3 Exercices



2.1 Qu'est-ce que Javascript ?

Historiquement, Javascript est un langage de programmation interprété, implémenté par les navigateurs Web pour la gestion de l'interactivité, le contrôle du navigateur, la modification dynamique du contenu affiché, et plus récemment la communication asynchrone avec le serveur (AJAX).



Contexte habituel de mise en oeuvre de Javascript

2.1.1 Contextes d'utilisation

Toutefois, Javascript est également utilisé dans d'autres contextes :

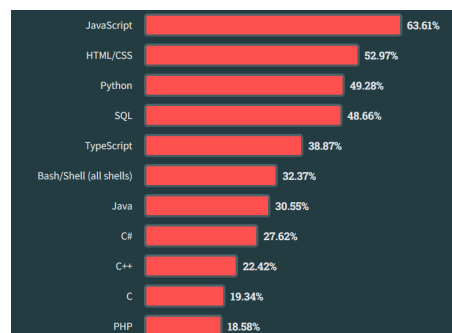
- pour la personnalisation de documents pdf, [\[Javascript for Acrobat\]](#)
- le développement de jeux en ligne, [\[Games by Brent Silby\]](#) [\[Instant adventure gaming\]](#)
- Adobe Flash et Flex, Unity : [ActionScript](#) est conforme à 100% avec EcmaScript 3,
- la diffusion en open source de l'interpréteur Javascript [V8 engine](#) en 2008, conçu pour Google Chrome a permis le développement de [Node.js](#).



[Node.js](#) est une plateforme logicielle orientée réseau, qui sert de base à de nombreux serveurs, sites, logiciels, et frameworks modernes. [\[Angular\]](#) [\[Atom\]](#) [\[Express\]](#) [\[Ionic\]](#) [\[Meteor\]](#)

En 2023, la plateforme Stackoverflow classe Javascript comme le langage le plus populaire depuis 11 années consécutives, devant HTML, CSS, Python, SQL et d'autres dont TypeScript, Java, C#, C++, C ou PHP. [\[stackoverflow langages\]](#)

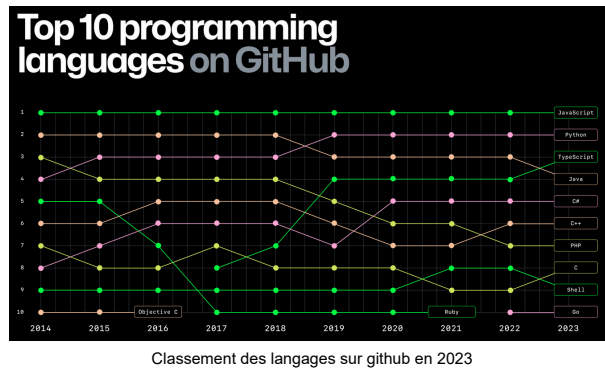
Javascript est par ailleurs le langage sur lequel sont basés les 6 premiers frameworks les plus utilisés. [\[stackoverflow frameworks\]](#)



Classement des langages sur stackoverflow en 2023



Javascript est également le langage le plus utilisé sur Github en 2023, depuis plus de dix années consécutives, suivi par Python et TypeScript. [\[github\]](#)



[\[Les langages les plus utilisés en 2023\]](#)

2.1.2 Javascript n'est pas Java

Pour des raisons marketing, lors de son lancement en décembre 1995 la version bêta de LiveScript pour Netscape Navigator 2.0 est rebaptisée Javascript.

Toutefois, et même si la syntaxe de Javascript est très similaire à celle de Java (*mais aussi de C ou de C++*) :

Javascript n'a strictement rien à voir avec Java !



2.1.3 Historique

[\[must read\]](#)

Mai à Décembre 1995

Mocha, ensuite renommé LiveScript, puis Javascript (*choix malheureux*) est développé en quelques semaines pour Netscape Navigator 2.0 à destination des designers (*et non de développeurs professionnels*) dans un contexte de concurrence avec les autres navigateurs.

[\[archive\]](#)

Août 1996

Javascript prend rapidement de l'importance. L'implémentation de Microsoft pour IE3 s'appelle JScript. [\[JScript vs. ECMA\]](#). L'interpréteur Javascript de Netscape (*initialement Mocha*) est réécrit et nommé SpiderMonkey.

Novembre 1996

Javascript est proposé à ECMA International pour devenir un standard, à l'origine des spécifications « ECMAScript » dont la première version a été adoptée en juin 1997 [\[ECMAScript 1\]](#).

Juin 1998

ECMAScript 2 s'aligne avec la version standardisée ISO/IEC 16262, sans introduire de modifications du langage



Décembre 1999

Standardisation d'ECMAScript 3 (*ES3 pour les intimes*) qui introduit notamment le traitement des expressions régulières et la possibilité de gérer les erreurs. [\[ECMAScript 3\]](#)

2000+

Après les débuts difficiles de Javascript (*imperfections suite aux contraintes de temps lors de sa conception, implémentations partielles, incompatibilités entre navigateurs*), ES3 est la première version raisonnablement supportée par la majorité des navigateurs de l'époque. [\[article\]](#)

AJAX (*Asynchronous Javascript And XML*) est une technique basée sur un composant XMLHttpRequest introduit en 1999 par Microsoft avec le navigateur IE5, reprise par les autres navigateurs sous la forme d'un objet Javascript et faisant l'objet de spécifications en soi. [\[AJAX\]](#) [\[XMLHttpRequest\]](#)

La popularisation de Javascript attire finalement des programmeurs professionnels et amène le développement des premières bibliothèques et frameworks, une amélioration des pratiques et une meilleure connaissance des possibilités et des spécificités du langage. [\[Prototype\]](#) [\[Dojo\]](#) [\[jQuery\]](#)

Côté spécifications, les divers acteurs (*Adobe, Mozilla, Opera, Microsoft, Yahoo*) ne parviennent pas à s'accorder sur les fonctionnalités à intégrer à ES4. Le travail s'arrête en 2003, puis reprend de 2005 à 2008 pour finalement abandonner ES4 au profit d'EcmaScript 3.1 renommé ES5 en 2009. [\[ES4 story\]](#)

2009

ES5 est sans doute la version la plus généralement supportée de Javascript. Elle apporte de nouvelles fonctionnalités par rapport à ES3, mais pas de modifications de syntaxe. On voit notamment apparaître JSON et le mode strict. [\[ECMAScript 5\]](#)

2011

EcmaScript 5.1 précise quelques ambiguïtés mais n'apporte pas de nouvelles fonctionnalités.

2015

ES6 standardisé sous l'appellation EcmaScript 2015 incorpore de nombreuses fonctionnalités provenant des discussions autour d'ES4, dont certaines introduisant de nouvelles syntaxes : (*let, const, fonctions flèche, classes, promesses, générateurs, itérateurs, opérateur reste, opérateur de propagation, déstructuration, modules...*). [\[fonctionnalités ES6\]](#)

Cette version est en général bien supportée par les navigateurs, par node.js et par les transpileurs (*transpilers*) capables de traduire du code ES6 en ES5. [\[compatibilité ES6\]](#) [\[Babel\]](#)

2016

ES7 est une petite mise à jour avec uniquement deux nouvelles fonctionnalités, dont l'opérateur d'exponentiation.



2.2 Démarrage rapide

2.2.1 Outils de développement

Même s'il est a priori possible de développer du code Javascript avec un simple éditeur de texte comme :



[Notepad]



[notepad++]



[Vim]



[Sublime Text]



[Atom]

on préférera utiliser un environnement de développement intégré (IDE) :



[Webstorm]



[Visual Studio]

[9 Best JS IDEs]

📧 Une adresse email en ec-lyon.fr vous donne droit à la licence étudiant gratuite pour WebStorm et les autres logiciels Jet Brains. [\[Licence étudiant\]](#)

Pour afficher les interfaces, exécuter les programmes, visualiser les résultats et déboguer le code on aura recours à un navigateur régulièrement mis à jour. Pour leurs "outils du développeur" on préférera Firefox ou Chrome.

Pour le développement d'un POC (*Proof Of Concept*) il ne sera en général pas nécessaire d'être compatible avec plusieurs navigateurs.

Pour un produit plus fini par contre (*MVP ou plus*), il faudra valider en phase de conception les fonctionnalités utilisées [\[Can I Use\]](#), et tester l'application en fonction de tous les navigateurs cibles (*en tenant compte des versions et des plateformes*).



[Firefox]



[Chrome]



[Opera]



[Safari]



[Edge]

2.2.2 Les outils du navigateur

La plupart des navigateurs (*sauf tablettes/smartphone*) possèdent des outils pour développeurs permettant d'examiner le code HTML, CSS et Javascript.

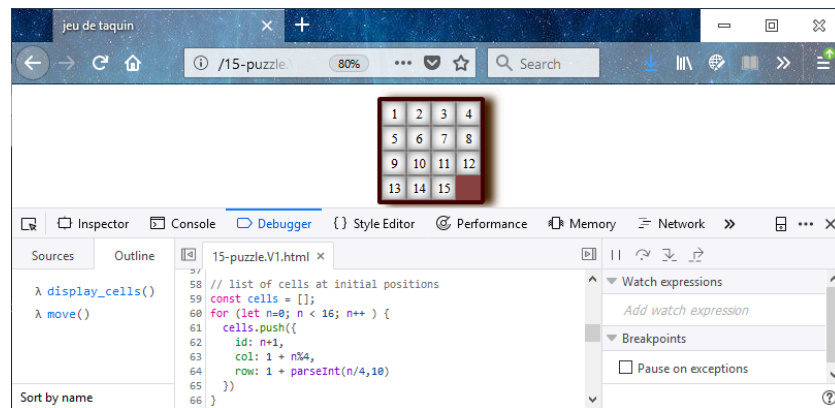


Concernant plus particulièrement Javascript, la console permet :

- d'exécuter du code simple via la ligne de commande,
- de visualiser et d'examiner le code de la page courante,
- d'afficher des messages (*très utile pour tracer l'exécution de code asynchrone*),
- de découvrir les erreurs d'exécution.

Le débogueur permet par ailleurs (*et entre autres*) :

- de positionner des points d'arrêt,
- d'exécuter le code pas à pas,
- d'examiner les requêtes réseau et la réponse obtenue (*cf. AJAX*).



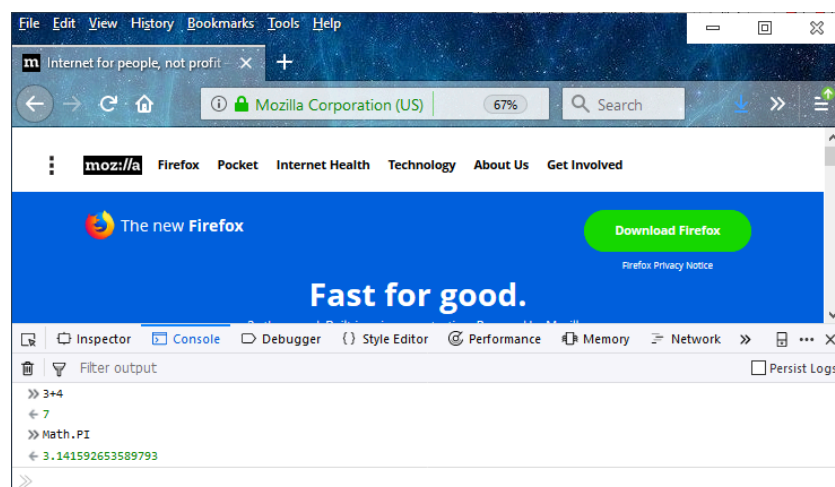
Navigateur avec débogueur

2.2.3 Exercice : prise en main de la console

Démarrer la console dans votre navigateur préféré :

- Firefox :** menu → "Développeur" → "Console Web"
- Chrome :** menu → "Outils" → "Outils du développeur" → Onglet "Console"
- Edge :** menu → "Outils de développement" → Onglet "Console"

Utiliser la ligne de commande pour exécuter quelques instructions...



Console javascript avec quelques commandes



2.2.4 HTML et la balise script

Pour faire exécuter au navigateur un programme Javascript plus conséquent, il suffit de créer un document avec une extension `.html` (cf. *exemple1.html*) :

```
<!DOCTYPE html>

<title>Exemple 1</title>
<meta charset="utf-8">
<div id="main">Mon premier programme Javascript</div>
<script>

// Insérer le code javascript ici

</script>
```

[exemple 0]

☞ Tous les exemples donnés dans la suite du cours pourront être implémentés en éditant ce fichier pour insérer le code source dans la zone indiquée. Pour tester le code il suffira d'ouvrir (*ou de recharger*) le fichier dans un navigateur.

On verra dans la suite comment afficher de différentes manières le fameux message "hello, world" popularisé par les inventeurs du langage C*.

* Brian W. Kernighan, Dennis M. Ritchie, "The C Programming Language", Prentice Hall Software Series, 1978 - section 1.1 "Getting started" [\[consulter\]](#)

La balise `<script>` permet également d'inclure du code provenant d'un document séparé :

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

☞ Attention : pour des raisons de sécurité et de confidentialité les navigateurs ne chargent pas de ressources extérieures (*url absolue*) depuis un document local (*avec une url en file: dans la barre d'adresse du navigateur*).

N.B. Le code de la balise `<script>` est exécuté par le navigateur lors du traitement séquentiel des éléments du document. Il faut donc faire attention à ne pas tenter de manipuler (*via des fonctions du DOM*) des éléments déclarés **après**.

Pour cette raison on préfère souvent positionner les scripts en fin de document :

```
<!DOCTYPE html>
<head>...</head>
<body>
...
<script>...</script>
</body>
```

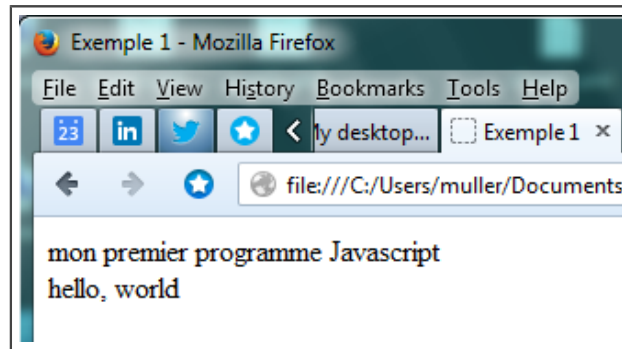
2.2.5 Afficher du texte avec `document.write`

Compléter le premier exemple en insérant le code ci-dessous à l'endroit adéquat :

```
document.write('hello, world');
```



[exemple 1]




Résultat d'exécution du programme exemple 1

Essayer avec plusieurs lignes consécutives affichant des messages différents :

```
document.write('hello, world<br>');  
document.write('hello, world again');
```

A noter : le texte affiché peut contenir des balises HTML.

 Bon à savoir : document.write ne fonctionne pas dans des documents XHTML.

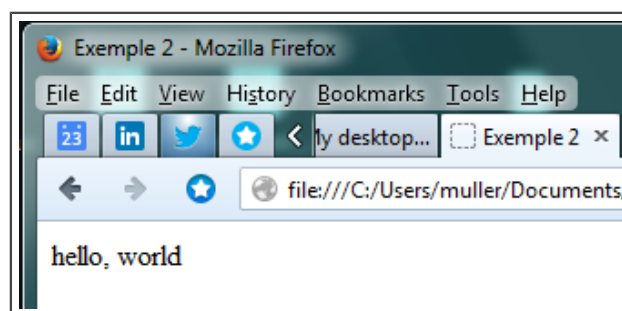
Par ailleurs, bien qu'utile pour du déverminage rapide, il n'est en général pas pertinent d'utiliser cette instruction dans du code opérationnel.

2.2.6 Modifier l'interface avec element.innerHTML

Créer maintenant un document nommé exemple2.html :

```
<!DOCTYPE html>  
  
<title>Exemple 2</title>  
<meta charset="utf-8">  
<div id="main">Mon second programme Javascript</div>  
<script>  
  
main.innerHTML = 'hello, world';  
  
</script>
```

[exemple 2]



Résultat d'exécution du programme exemple 2



Remarquer comment le texte initial "Mon second programme Javascript" a été remplacé (*et non complété, comme précédemment*) par le message "hello, world".

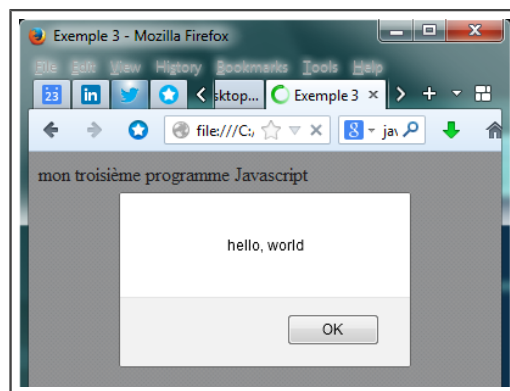
✎ Bon à savoir : l'élément "main" qui apparaît dans l'expression `main.innerHTML` du programme Javascript désigne l'élément `div` qui porte l'attribut `id="main"`. C'est une spécificité HTML5.

2.2.7 Afficher dans un popup avec alert

Développer maintenant un document `exemple3.html`, avec le code :

```
alert('hello, world');
```

[exemple 3]



Résultat d'exécution du programme exemple 3

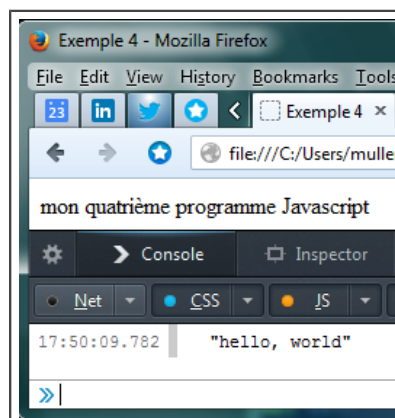
✎ Bon à savoir : la fenêtre popup instanciée par la fonction `alert()` est dite "modale", car l'utilisateur ne peut avoir aucune interaction avec la page, tant qu'il n'a pas actionné le bouton "Ok". Pour cette raison, il faut éviter d'utiliser cette fonction de manière trop répétée.

2.2.8 Les méthodes de la console

Lors du développement d'un programme on désire souvent afficher des messages de service uniquement destinés au développeur. Ces messages ne doivent pas apparaître à l'utilisateur final. Ils peuvent être dirigés vers la console :

```
console.log('hello, world');
```

[exemple 4]



Résultat d'exécution du programme exemple 4



A noter : il ne faut pas oublier d'invoquer la console pour voir le message...

👉 Bon à savoir : il est possible d'afficher d'autres types de messages dans la console à l'aide des fonctions `console.error()`, `console.info()` et `console.warn()`. [\[API console\]](#).

2.2.9 Exercice : affichage d'information

Développer un programme qui :

- 1 affiche un message de départ dans la console,
- 2 demande l'attention de l'utilisateur à l'aide d'une alerte par popup,
- 3 affiche un message de confirmation dans la console lorsque l'utilisateur a répondu "Ok",
- 4 modifie le contenu de la page du navigateur,
- 5 affiche "travail effectué" à l'aide d'un popup,
- 6 puis confirme la fin des opérations dans la console.



2.3 Caractéristiques générales du langage

2.3.1 Structure du code

JavaScript permet la programmation impérative, structurée avec une syntaxe héritée de Java qui hérite lui-même du langage C.

La syntaxe des opérateurs (*arithmétiques, logiques, binaires, de comparaison, d'incrément/décroissement, d'affectation, ternaire ?:*) et des instructions (*if, else, for, while, switch, break, continue, ...*) est classique.

A noter que l'opérateur « + » est également utilisé pour la concaténation de chaînes de caractères...

```
<div id="my_div"></div>
<script>
text = "";
for ( i = 0; i < 25; i++ ) {
    text += i + " ";
}
my_div.innerHTML = text;
</script>
```

☞ Un temps d'adaptation sera nécessaire pour ceux qui ne connaissent que Python.

2.3.2 Point virgule

Le point-virgule « ; » (*semicolon*) peut être omis en fin d'instruction ... ou pas !

Il est souvent conseillé de terminer systématiquement toutes les instructions par un point-virgule, mais certains préconisent de *ne pas* en mettre... [au choix] [optionnel]

Aucune des approches ne résout tous les problèmes, car Javascript termine automatiquement certaines instructions (*qu'il est nécessaire de connaître*) comme `return`, à l'aide d'un mécanisme connu sous l'acronyme **ASI** (*Automatic Semicolon insertion*) :

```
return
a+b;
```

est interprété comme :

```
return;
a+b;
```


☞ Conseil : ne jamais laisser l'instruction `return` seule sur une ligne (*ainsi que quelques autres comme `let`, `const`, `import`, `export`, `continue`, `break`, ou `throw`*).

Par ailleurs, en l'absence de « ; » Javascript termine automatiquement une instruction si la prise en compte du caractère suivant provoquerait une erreur. Ceci conduit à une autre famille de cas pathologiques qu'il faut connaître :

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

provoque une erreur **c is not a function...**



 **Conseil** : Ne jamais commencer une instruction par une parenthèse ouvrante « (», crochet ouvrant « [», un signe « / », « + », ou « - » sans « ; » entre cette instruction et la précédente.

2.3.3 Définition de fonctions

Une **déclaration de fonction** permet de définir une fonction avec la liste de ses paramètres.
[\[déclaration de fonction\]](#)

Les déclarations de fonctions sont automatiquement hissées (*hoisted*) au début du contexte global (*ou de la fonction parent*). Ceci permet d'utiliser une fonction avant sa définition :

```
console.log(factorielle(20));  
...  
function factorielle(n) {  
    if ( n % 1 ) return undefined; // n non entier  
    return ( n ? n * factorielle(n-1) : 1 );  
}
```

N.B. Pour renvoyer une valeur, une fonction doit obligatoirement comporter une instruction `return`.

Une **expression de fonction** (*function expression*) affecte une **fonction anonyme** (*i.e. sans nom*) à une variable. Elle s'utilise comme la précédente : [\[expression de fonction\]](#)

```
sum = function(a,b) {  
    return a+b;  
};  
console.log(sum(33,99));
```

N.B. Les fonctions définies de cette manière ne sont pas hissées (*hoisted*) et ne peuvent donc pas être utilisées avant leur affectation.

Lorsque la fonction doit faire référence à elle-même (*fonction récursive par exemple*), une fonction anonyme ne convient pas.

On parle alors d'une **expression de fonction nommée** (*named function expression*). Le nom de la fonction est uniquement connu à l'intérieur du corps de celle-ci :

```
factorielle = function fact(n) {  
    if ( n % 1 ) return undefined; // n non entier  
    return ( n ? n * fact(n-1) : 1 );  
}  
console.log(factorielle(20));
```

ES6 a introduit une syntaxe plus légère pour la définition de fonctions à l'aide d'une expression : les **fonctions flèche** (*arrow functions*). Leur nom provient de la nouvelle syntaxe : [\[fonctions flèche\]](#)

```
sum = (a,b) => a+b; // équivalent à sum = (a,b) => { return a+b; };  
console.log(sum(4,5));
```

Lorsqu'il n'y a qu'un seul paramètre, les parenthèses sont optionnelles :

```
sq = x => x*x; // équivalent à sq = (x) => { return x*x; };  
console.log(sq(8));
```

N.B. Les fonctions flèche ont d'autres spécificités qui seront abordées plus loin.



Les fonctions JavaScript sont **variadiques** c'est-à-dire que le nombre de paramètres passés lors de l'appel peut ne pas correspondre à celui de la déclaration :

```
function factorielle(n) {  
    if ( n % 1 ) return undefined; // n non entier  
    return ( n ? n * factorielle(n-1) : 1 );  
}  
console.log( factorielle() ); // n est undefined → renvoie 1
```

Pour accéder aux paramètres éventuellement surnuméraires, on peut utiliser le tableau `arguments` automatiquement présent dans toute fonction :

```
function add() {  
    s = 0;  
    for ( n = 0; n < arguments.length; n++ ) {  
        s += arguments[n];  
    }  
    return s;  
}  
console.log( add(1,2) ); // 3  
console.log( add(1,2,3) ); // 6
```

2.3.4 Portée des variables

En Javascript on peut ne pas déclarer une variable avant de lui affecter une valeur :

```
var x = 1; // variable déclarée et initialisée  
var y; // variable déclarée mais non initialisée  
z = 2; // création automatique avec affectation de valeur  
console.log(x); // 1  
console.log(y); // undefined  
console.log(z); // 2  
console.log(t); // ReferenceError: t is not defined
```

Par défaut, **les variables non déclarées sont globales !**

```
x = 33; // x global  
function add(v) {  
    return (x += v); // fait référence à x global  
}  
console.log(x); // 33  
console.log( add(1) ); // 34  
console.log( add(1) ); // 35  
console.log(x); // 35
```

```
function sum(v) {  
    var y = 0; // x local  
    return (y += v);  
}  
console.log( sum(1) ) // 1  
console.log( sum(1) ) // 1  
console.log(y); // ReferenceError: y is not defined
```

Ce point autorise la création (ou la modification), accidentelle ou pas, de variables globales depuis n'importe quelle fonction, éventuellement située aux fins fonds de n'importe quelle bibliothèque



utilisée par le code. Il est couramment admis que cet aspect fait partie du côté obscur de Javascript...
[\[article 2012\]](#)

Le mode strict introduit avec ES5 interdit notamment la création accidentelle (*ou volontaire*) de variables globales, mais pas leur modification : [\[Strict mode\]](#)

```
var y = 1;
function bad() {
  x = 33; // création d'une variable globale
  y = 33; // modification d'une variable globale
}
bad();
console.log(x); // 33
console.log(y); // 33
```

```
'use strict';
var y = 1;
function bad() {
  x = 33; // x n'est pas déclarée...
  y = 33; // y est globale
}
bad(); // ReferenceError: assignment to undeclared variable x
```

☞ La modification de variables globales ne peut être détectée que par l'examen du code à l'aide d'un analyseur de code (*linter*). [\[ESLint\]](#) [\[JSHint\]](#) [\[TSLint\]](#)

Une fois **déclarées**, la portée des variables est limitée à **la fonction** (*et non pas le bloc*) dans laquelle elles l'ont été.

```
function f() {
  console.log(n); // undefined
  for (var n = 1; n < 10; n++){};
  console.log(n); // 10
}
f();
```

Dans l'exemple ci-dessus la variable `n` est connue avant d'être déclarée, mais non initialisée. En effet, la **déclaration** des variables à l'aide du mot-clé `var` est hissée (*hoisted*) en début de la fonction, mais les variables prennent uniquement une valeur lorsque la ligne de code qui les initialise est exécutée.

Sa valeur est par ailleurs accessible en-dehors du bloc `for` au sein duquel elle a été déclarée.
[\[hoisting\]](#)

ES6 introduit un nouveau mot-clé `let` qui déclare une variable dont la portée est limitée au **bloc** (*et non pas à la fonction*) et dont la déclaration n'est pas hissée. [\[let\]](#)

```
function f() {
  console.log(x) // ReferenceError: can't access x before initialization
  let x = 1;
  for (let n = 1; n < 10; n++) {}
  console.log(n); // ReferenceError: n is not defined
}
f();
```




2.3.5 Protection de l'espace global

Pour éviter de "polluer" l'espace global, il existe un motif (*design pattern*) connu sous l'acronyme IIFE (*Immediately Invoked Function Expression*) très utilisé par de nombreux développeurs. [IIFE]

Il consiste à définir une fonction anonyme, n'utilisant que des variables locales, et à l'appeler immédiatement :

```
y = (function () {  
    var x = 1; // variable inaccessible depuis l'extérieur du bloc  
    return x+2;  
})(); // exécution immédiate de la fonction...  
console.log(y); // 3 ... son résultat est accessible  
console.log(x); // ReferenceError: x is not defined
```

☞ Ce pattern est très largement utilisé au sein de nombreux frameworks.

ES6 a également introduit le mot-clé `const` pour déclarer des variables dont il n'est pas prévu de modifier la valeur (*i.e. des constantes*). Ceci est très utile pour prévenir la modification accidentelle de constantes globales par du code inclus. [const]

```
const cours = "fullstack"  
function f() {  
    cours = "XML";  
}  
f(); // TypeError: invalid assignment to const 'cours'
```

☞ Comme pour `let` cette déclaration n'est pas hissée.

2.3.6 Types primitifs

Javascript possède 6 types primitifs plus `Object`. [types de données]

Boolean

Représentations littérales : `true` et `false`.

Ce type est notamment requis par le contexte de l'instruction `if` et comme premier argument de l'opérateur triadique « ? » :

```
if ( x == false ) { console.log("ok"); }
```

undefined

Une seule valeur possible : `undefined`.

Correspond à la valeur des variables non initialisées.

```
var x; // variable déclarée, valant undefined
```

Null

Une seule valeur possible : `null`.

```
var x = null; // variable initialisée, valant null
```

☞ Souvent utilisé comme valeur de retour par une fonction renvoyant un objet, lorsqu'il n'y a pas d'objet à renvoyer...



Number

Ce type correspond à la représentation normalisée IEEE 754 double précision.

Il n'y a pas d'entiers en JavaScript : tous les nombres sont traités en virgule flottante. [\[IEEE 754\]](#)

Exemples de représentations littérales :

```
i = 314;           // un entier ... codé en virgule flottante
f = 3.14;          // notation classique
x = 3.14e2;        // notation exponentielle, égal à 314
y = 0377;          // notation octale, égal à 255
z = 0xff;          // notation hexa (MAJ. ou min.), égal à 255
a = Infinity;      // positif ou négatif, cf. IEEE 754
b = NaN;           // cf. IEEE754
```

Attention : la représentation littérale d'un nombre avec des caractères « 0 » initiaux est interprétée comme étant de l'octal !

☞ Le mode strict (cf. ES5) modifie la représentation littérale des valeurs octales pour éviter les erreurs involontaires :

```
'use strict';
var x = 0o377;      // nouvelle notation
console.log(x);     // 255
var y = 0377;       // SyntaxError
```

String

Représente une séquence d'unités de code 16 bits (*Unicode*).

Exemples de représentations littérales :

```
s = "Hello !";      // guillemets doubles
t = 'Ok';            // guillemets simples
a = '<a href="#">ici</a>'; // ok
z = "<a href='#'>ici</a>"; // syntax error
b = "<a href=\"#\">ici</a>"; // ok
```

Il existe deux manières pour accéder à un caractère particulier d'une chaîne :

```
var h = "Hello !";
console.log( h.charAt(6) ); // méthode historique, affiche !
console.log( h[6] );       // ES5, même résultat !
console.log( h.length );   // 7, nombre de caractères de la chaîne
```

N.B. En Javascript les chaînes de caractères sont immuables (*immutable*). On ne peut pas modifier individuellement un caractère particulier d'une chaîne :

```
var h = "Hello !";
h[6] = "?";           // rate en silence
console.log( h );     // Hello !
```

```
'use strict';
var h = "Hello !";
h[6] = "?";           // TypeError: 6 is read-only
```

N.B. En Unicode certains caractères composés ou issus des plans astraux sont représentés avec deux unités de 16 bits au lieu d'une seule.



Il faut se méfier de ces cas particuliers, car les méthodes `charAt`, `length` et l'opérateur `[]` travaillent sur la séquence d'unités de 16 bits (*codepoints*) et non pas sur les caractères ou glyphes visibles par les humains : [\[JS et Unicode\]](#)

```
var cool = "\u{1F60E}";  
console.log(cool, cool.length, cool[0]); // # 2 ☐  
  
var drink = 'cafe\u0301';  
console.log(drink, drink.length, drink[3]); // => café, 5, e
```

ES6 permet de résoudre le problème des caractères composés :

```
str = "J'aime le cafe\u0301"; // 14 caractères, 15 points  
console.log( str.length ); // 15  
console.log( str.normalize() ); // J'aime le café  
console.log( str.normalize().length ); // 14
```

Pour pouvoir itérer sur tous les **caractères** d'une chaîne et non pas sur les unités de code, il est nécessaire de créer un tableau avec la liste des caractères.

Avec ES6 (*encore*) et l'opérateur de propagation (*spread*) :

```
str = "J'aime le cafe\u0301 \u{1F642}"; // 16 caractères, 18 points  
console.log(str.length); // 18  
console.log([...str].length); // 17  
console.log([...str.normalize()].length); // 16
```

Symbol

Symbol est un type primitif à valeurs uniques et immuables introduit par ES6, essentiellement utilisé comme clé pour les attributs protégés d'un objet. [\[Symbol\]](#)

2.3.7 Typage dynamique

Javascript est un langage faiblement typé. Un type est associé à une valeur, non à une variable. Ainsi, la valeur d'une même variable peut changer de type au cours du temps :

```
s = 1; // la valeur de s est du type number  
s = "Ok"; // la valeur de s est du type string
```

Lors de l'évaluation d'une expression, JavaScript effectue des conversions automatiques, pour obtenir à chaque fois le type requis par un contexte donné :

```
s = 1;  
x = ( s ? 1 : 0 ); // la valeur de s est convertie en booléen  
  
d = "33 km";  
y = d.substring(0,2) - 10;  
console.log(typeof y, y); // number 23  
  
z = d.substring(0,2) + 10;  
console.log(typeof z, z); // string 3310 :-(  
  
t = +d.substring(0,2) + 10;  
console.log(typeof t, t); // number 43 :-)
```



☞ Si certains apprécient cet aspect du langage (*programmeurs ?*) d'autres l'ont en horreur (*chefs de projet ?*). C'est l'une des raisons pour l'apparition du langage **TypeScript** utilisé notamment par le framework **Angular**.

L'opérateur typeof

L'opérateur `typeof` permet a priori de déterminer le type de la valeur d'une variable. Les valeurs possibles sont "undefined", "boolean", "number", "string", "object", ou "function" :

```
var x;
console.log(typeof x); // undefined
x = null;
console.log(typeof x); // object (!)
x = true;
console.log(typeof x); // boolean
x = 1;
console.log(typeof x); // number
x = 'Ok';
console.log(typeof x); // string
x = Date;
console.log(typeof x); // function
x = new Date();
console.log(typeof x); // object
x = [1,2];
console.log(typeof x); // object
```

N.B. `typeof null` renvoie "object". Encore un aspect du côté obscur de Javascript, très critiqué, conservé pour compatibilité ascendante, et dû à la période de conception très réduite...

2.3.8 Conversions de type

Il est parfois utile de contraindre une valeur pour appartenir à un type précis. Voici comment procéder pour obtenir chacun des types primitifs :

undefined

Pour s'assurer qu'une variable soit du type undefined, il faut lui affecter la valeur `undefined` :

```
x = undefined;
console.log(typeof x, x); // undefined undefined
```

null

De même pour forcer une variable à appartenir au type null, il faut lui affecter la valeur `null` :

```
x = null;
console.log(typeof x, x); // object null
```

boolean

Pour convertir une variable en booléen, quel que soit son type initial :

```
x = (x ? true : false); // true boolean
```

☞ Les valeurs 0, -0, null, NaN, undefined, et "" donnent false, tout le reste donne true, y compris les chaînes "0", "0.0" et "false" et n'importe quel objet (sauf null).



Pour éviter les conversions implicites lors d'un test d'égalité, il faut utiliser les opérateurs « === » et « !== » :

```
t = true;
x = 1;

// x est égal à t
console.log("t " + (( t == x ) ? "est" : "n'est pas") + " égal à t");

// x n'est pas strictement égal à t
console.log("t "+(( t === x ) ? "est" : "n'est pas")+" strictement égal à t");
```

string

Pour obtenir une chaîne de caractères, le plus simple est de concaténer avec la chaîne vide :

```
function log(s) { console.log(typeof s, s); };
var x;           log(x+''); //string undefined
x = null;        log(x+''); //string null
x = true;        log(x+''); //string true
x = 1;           log(x+''); //string 1
x = 'Ok';        log(x+''); //string Ok
x = Date;        log(x+''); //string function Date() { [native code] }
x = new Date();  log(x+''); //string Thu Aug 23 2018 14:44:43 GMT+0200
x = [1,2];       log(x+''); //string 1,2
```

number

Pour obtenir un nombre, on peut requérir à l'opérateur monadique « + » pour effectuer une conversion implicite :

```
function log(s) { console.log(typeof s, s); };
var x;           log(+x); // number NaN
x = null;        log(+x); // number 0
x = true;        log(+x); // number 1
x = 1;           log(+x); // number 1
x = '23';        log(+x); // number 23
x = '23km';      log(+x); // number NaN
x = Date;        log(+x); // number NaN
x = new Date();  log(+x); // number 1535029702121
x = [1,2];       log(+x); // number NaN
```

Dans le cas d'une chaîne dont seuls les premiers caractères correspondent à un nombre, on peut requérir à la fonction globale `parseInt()` :

```
x = '23';        console.log(parseInt(x)); // 23
x = '23km';      console.log(parseInt(x)); // 23
x = 3.141592;    console.log(parseInt(x)); // 3
```

⚠ Attention : `parseInt` admet un second paramètre correspondant à la base dans laquelle le nombre est exprimé. Lorsque la chaîne commence par le chiffre « 0 », la base par défaut a changé au cours de l'histoire : 8 avant ES5, 10 après...

```
x = '0377';
console.log(parseInt(x)); // 255 avant ES5, 377 après ...
console.log(parseInt(x,10)); // 377
```

Conclusion : toujours préciser la base (i.e. 10) lors d'un appel à `parseInt()`.



De même, il est possible de convertir une chaîne en nombre non entier avec la fonction `parseFloat()` :

```
x = "33.5km";  
console.log(parseFloat(x)); // 33.5  
  
x = "381.5e3km";  
console.log(parseFloat(x)); // 381500
```

🔑 `parseFloat` travaille toujours en base 10...

2.3.9 Tableaux

Un tableau peut être initialisé de manière littérale :

```
semaine = [ "lundi", "mardi", "mercredi", "jeudi",  
            "vendredi", "samedi", "dimanche" ];
```

Pour accéder à un élément de tableau, on utilise classiquement l'opérateur « `[]` » avec un indice entier qui démarre à 0 :

```
for ( n = 0; n < semaine.length; n++ ) {  
    console.log(semaine[n]);  
}
```

🔑 Noter `semaine.length`...

Sauf nécessité d'optimisation, il est inutile de réserver la taille d'un tableau lors de sa déclaration, puisqu'elle est automatiquement étendue en cas de besoin :

```
a = [];  
for ( n = 0; n < 100; n++ ) { a[n] = n*n; };
```

Les éléments d'un tableau peuvent être tous de type différent :

```
a = [];  
a[0] = 1; a[2] = 'ok'; a[3] = [1,2,3]; // ...
```

2.3.10 Exercice : prise en main de Javascript

Pour les matheux...

Dans une page html et à l'aide de Javascript :

- 1 Calculer et afficher les 40 premiers nombres de Fibonacci. [\[réf. Wikipédia\]](#)
- 2 Calculer et afficher les nombres premiers inférieurs à 1024. Combien cela en fait-il ? [\[réf. Wikipédia\]](#)
- 3 Trouver les 5 premières décimales de π avec la formule de Leibniz-Gregory. Combien faut-il d'itérations ? [\[réf. en ligne\]](#)
- 4 Combien faut-il d'itérations pour trouver les 5 premières décimales de π avec la formule de Nilakantha ? [\[réf. en ligne\]](#)
- 5 Toujours avec cette dernière formule, trouver les 13 premières décimales. Combien faut-il d'itérations ?



2.4 Orientation Objet

2.4.1 Tableaux associatifs

Contrairement aux tableaux à indices entiers (*cf. listes en Python*) les tableaux associatifs (*cf. dictionnaires en Python*) utilisent des chaînes de caractères (*i.e. des clés*) pour accéder à leur contenu. Leur expression littérale est très classique :

```
parents = { papa : "Raymond", maman : "Ginette" };
```

On accède à un élément particulier en précisant la clé à l'aide l'opérateur « [] » :

```
console.log(parents['papa']); // Raymond
```

Il faut une instruction spéciale pour boucler sur toutes les clés d'un tel tableau :

```
for ( p in parents ) {  
    console.log(p, parents[p]);  
}
```

N.B. Là encore, les éléments d'un tableau associatif peuvent être de types différents.

L'opérateur .

Lorsque la valeur de la clé correspond à un identifiant (*pas d'espace, ne commence pas par un chiffre, ...*), il existe un opérateur alternatif (*syntactic sugar*) équivalent à « [] » :

```
parents = { papa : "Raymond", maman : "Ginette" };  
p = parents["papa"]; // opérateur []  
m = parents.maman;   // opérateur .  
  
console.log(p);       // Raymond  
console.log(m);       // Ginette
```

Ces notations sont équivalentes et interchangeables.

N.B. La différence se situe dans le fait que l'opérateur [] admet une expression (*nom de variable ou expression plus complexe*), alors que . nécessite la valeur de la clé :

```
for ( p in parents ) {  
    console.log ( p + " : " + parents[p] );  
}
```

```
> papa : Raymond  
> maman : Ginette
```



2.4.2 JSON

L'écriture littérale des tableaux classiques et associatifs est à la base de la notation JSON (*JavaScript Object Notation*). JSON est souvent utilisée comme alternative à XML dans le cadre des services Webs et par les applications pour l'échange d'information entre front et back :

```
{
  "papa" : { "nom": "Deubaze", "prénom": "Raymond" },
  "maman" : { "nom": "Ringard", "prénom": "Ginette" },
  "enfants" : [ "Jean", "caroline", "zoé" ],
  "animaux" : [
    { "nom": "Minou", "type": "chat" },
    { "nom": "Médor", "type": "chien" }
  ]
  "revenu": 54.376
}
```

☞ JSON impose que le nom des clés soit spécifié entre quotes "", ce qui n'est pas le cas de JavaScript.

N.B. Malgré la signification de l'acronyme, l'usage de JSON n'est pas limité à JavaScript. Il existe des encodeurs et décodeurs dans pratiquement tous les environnements imaginables (ASP, C, C++, C#, Java, Objective C, Perl, PHP, Python, Ruby, Tcl, VB, ...).

2.4.3 Objets

En JavaScript il n'y a aucune différence entre objets et tableaux associatifs.

Un élément de tableau associatif est un attribut de cet objet. Quel que soit l'angle d'approche les deux sont équivalents, et les opérateurs "interchangeables".

☞ On a vu qu'une fonction peut être affectée à une variable à l'aide d'une expression de fonction. Lorsqu'un attribut d'un objet est une fonction, on appelle celui-ci une méthode...

```
parents = {
  papa : "Raymond",
  maman : "Ginette",
  hello : function() { return 'bonjour !'; }
};
console.log(parents.hello()); // bonjour !
```

... ainsi *console* est un objet, et *log* une de ses méthodes...

Les objets disponibles dans un programme Javascript peuvent être :

- des objets natifs standards définis par le langage, [objets natifs]
- fournis par l'application hôte (*le navigateur ou node.js*), [APIs Web] [API node.js]
- créés par le programme utilisateur.



Les attributs correspondent aux valeurs particulières de l'implémentation IEEE754 (*plus petit et plus grand nombre, entier, NaN, infini*).

```
console.log(Number.MIN_VALUE); // 5e-324
console.log(Number.MAX_VALUE); // 1.7976931348623157e+308
```

Les méthodes de classe apparues avec ES6 testent une valeur par rapport à ces constantes :

```
check = x => x+(Number.isInteger(x) ? " est":" n'est pas")+ ' entier';
console.log( check(5) ); // 5 est entier
console.log( check(5/2) ); // 2.5 n'est pas entier
```

Les méthodes d'instance (*i.e. applicables à une valeur*) servent à obtenir diverses représentations du nombre, et renvoient une chaîne de caractères :

```
n = 3141.5926535;
console.log( n.toExponential(3) ); // 3.142e+3
console.log( n.toFixed(1) ); // 3141.6
console.log( n.toLocaleString('fr-FR') ); // 3 141,593
console.log( n.toPrecision(4) ); // 3142
console.log( n.toString(16) ); // c45.97b823c85c
```

Math

L'objet Math sert d'API pour les constantes et les fonctions mathématiques. [\[Math\]](#)

Le nom des attributs permet assez intuitivement de remonter à leur signification :

- E, LN10, LN2, LOG2E, LOG10E, PI, SQRT1_2, SQRT2.

```
console.log(Math.PI); // 3.141592653589793
```

Il dispose par ailleurs des méthodes suivantes (** introduites par ES6*) :

- *min()*, *max()*, *abs()*, *random()*, *ceil()*, *floor()*, *round()*, *sign()**, *trunc()**
- *exp()*, *log()*, *pow()*, *sqrt()*, *cbrt()**, *expm1()**, *log1p()**, *hypot()**
- *sin()*, *cos()*, *tan()*, *asin()*, *acos()*, *atan()*, *atan2()*,
- *sinh()**, *cosh()**, *tanh()**, *asinh()**, *acosh()**, *atanh()**

dont le nom est parlant pour la plupart, sauf peut-être $cbrt(x) = x^{1/3}$, $expm1(x) = e^x - 1$, $log1p(x) = \log(1+x)$, $hypot(a_i) = \sqrt{\sum a_i^2}$.

📖 Les fonctions trigonométriques travaillent avec des angles en radians : *acos()* renvoie une valeur comprise entre 0 et π , *asin()* et *atan()* entre $-\pi/2$ et $+\pi/2$, *atan2()* entre $-\pi$ et $+\pi$.

```
console.log ( Math.PI == Math.acos(-1) ); // true
```



String

L'objet String encapsule une chaîne de caractères et possède de nombreuses méthodes utiles à la manipulation de chaînes. [String]

```
c = "une chaîne";  
s = String("une autre chaîne");  
o = new String("un objet");  
console.log(typeof c, c.length, c); // string 10 une chaîne  
console.log(typeof s, s.length, s); // string 16 une autre chaîne  
console.log(typeof o, o.length, o); // object 8 String { "un objet" }
```

Noter comment la chaîne `c` a été automatiquement convertie en objet pour pouvoir afficher son attribut `length`. D'autres conversions automatiques sont également à l'œuvre avec du code comme :

```
s = "hello" + ", ".concat("world");  
console.log(typeof s, s.length, s); // string 12 hello, world
```

☞ Déjà vu : l'attribut `length` donne le nombre de points unicode de la chaîne.

Les méthodes sont nombreuses (* introduites par ES6) :

charAt(), charCodeAt(), codePointAt(), concat(), endsWith(), includes(), indexOf(), lastIndexOf(), localeCompare(), match(), normalize(), padEnd(), padStart(), repeat(), replace(), search(), slice(), split(), startsWith(), substring(), toLocaleLowerCase(), toLocaleUpperCase(), toLowerCase(), toUpperCase(), trim(), trimStart(), trimLeft(), trimEnd(), trimRight()...

et laissées à la sagacité du lecteur.

JSON

L'objet JSON possède deux méthodes : [JSON]

JSON.stringify()

sert à sérialiser des objets Javascript pour les représenter sous la forme d'une chaîne de caractères au format JSON.

```
o = { a: true, b: 2, c: 'trois', d: [1,2,3,4] };  
j = JSON.stringify(o);  
  
console.log(j); // {"a":true,"b":2,"c":"trois","d":[1,2,3,4]}
```

JSON.parse()

effectue l'opération inverse en prenant une chaîne au format JSON et en renvoyant l'objet désérialisé.

```
p = JSON.parse(j);  
console.log(p); // Object {a: true, b: 2, c: "trois", d: (4) [...]}
```

☞ Comme déjà mentionné, la possibilité de sérialiser des objets natifs vers JSON puis de les désérialiser existe dans de nombreux langages. JSON est de ce fait devenu l'un des formats phares pour l'échange de données entre applications.



Array

L'objet Array encapsule un tableau (cf. *liste en Python*). Un tableau peut être initialisé avec une valeur littérale, ou via un appel au constructeur. [\[Array\]](#)

```
a = [1, 2, 3, 4];  
b = Array(1, 2, 3, 4)  
c = new Array(1, 2, 3, 4)  
console.log(typeof a, a.length, a); // object 4 Array(4) [1, 2, 3, 4]  
console.log(typeof b, b.length, b); // object 4 Array(4) [1, 2, 3, 4]  
console.log(typeof c, c.length, c); // object 4 Array(4) [1, 2, 3, 4]
```

L'attribut `length` est automatiquement mis à jour pour correspondre à l'index du dernier élément. Il peut être modifié. Si on lui affecte une valeur inférieure à sa valeur actuelle, le tableau est tronqué.

```
a = []; console.log(a.length); // 0  
a[99] = 1; console.log(a.length); // 100  
a.length = 50; console.log(a[99]); // undefined
```

Attention, en Javascript les tableaux (ainsi que les objets) sont manipulés **par référence** (cf. *pointeurs en C/C++*) :

```
a = [1, 2, 3, 4, 5];  
b = a;  
a[2] = 30;  
console.log(b); // Array(5) [ 1, 2, 30, 4, 5 ]
```

L'objet Array possède un grand nombre de méthodes qui renvoient des informations sur le contenu du tableau, permettent de modifier son contenu, ou d'itérer de diverses manières sur ses éléments.

Méthodes "de classe"

Array.from(), Array.isArray(), Array.of()**

```
s = 'hello, world';  
a = Array.from(s);  
console.log(typeof s, s); // string hello, world  
console.log(typeof a, a); // object Array(12) ["h","e","l","l","o", ...]
```

Accesseurs (renvoient des informations sans modifier le tableau)

*concat(), includes()**, indexOf(), join(), lastIndexOf(), slice(), toString(), toLocaleString()*

```
a = [1, 2, 3, 4];  
s = a.slice(1, 3).concat([10, 20]).join(' - '); // "2 - 3 - 10 - 20"
```

Modification du contenu du tableau

copyWithin(), fill()*, pop(), push(), reverse(), shift(), sort(), splice(), unshift()*

```
a = []; for (n=0; n < 5; n++) a[n] = +Math.random().toFixed(2);  
console.log(a); // [0.87, 0.33, 0.15, 0.86, 0.48]  
console.log(a.sort((x, y) => x - y)); // [0.15, 0.33, 0.48, 0.86, 0.87]
```



Méthodes d'itération sur le contenu (*attention aux tableaux creux !*)

*entries()**, *every()*, *filter()*, *find()**, *findIndex()**, *forEach()*, *keys()*, *map()*, *reduce()*, *reduceRight()*, *some()*, *values()*

```
a = []; a[99] = 1;
a.forEach((x,i) => console.log(i+' : '+x)); // 99 : 1
```

Date

L'objet Date permet de manipuler des dates. Il n'existe pas d'expression littérale pour créer une date, mais le constructeur accepte divers types d'attributs : [\[Date\]](#)

```
now = new Date(); console.log(now); // Date 2018-08-27T09:00:46.779Z
d1 = new Date(1535360330304); // Date 2018-08-27T08:58:50.304Z
d2 = new Date(2018,8,3,9); // Date 2018-09-03T07:00:00.000Z
```

N.B. En interne, JavaScript représente une date sous la forme du nombre de millisecondes écoulées depuis le 1er janvier 1970 à 0h 00 UTC.

Cet objet possède de nombreuses méthodes pour accéder aux différentes composantes d'une date ou les modifier individuellement :

setTime(t), *set*(x)*, *setUTC*(x)*, *getTime()*, *get*()*, *getUTC*()*, *getTimezoneOffset()*

... les méthodes qui comportent le terme UTC travaillent avec l'heure universelle (*GMT*), les autres avec l'heure locale. L'astérisque* peut être remplacé par les valeurs suivantes :

FullYear, *Month*, *Date*, *Day*, *Hours*, *Minutes*, *Seconds*, *Milliseconds*

```
d = new Date(); console.log(d); // Date 2018-08-27T10:00:33.334Z
d.setHours(14); console.log(d); // Date 2018-08-27T12:00:33.334Z
d.setUTCHours(14); console.log(d); // Date 2018-08-27T14:00:33.334Z
console.log(d.getFullYear()); // 2018
```

Les méthodes restantes servent à récupérer diverses représentations de la date sous forme de chaîne de caractères :

toString(), *toISOString()*, *toJSON()*, *toLocaleDateString()*, *toLocaleString()*,
toLocaleTimeString(), *toString()*, *TimeString()*, *UTCString()*

```
d = new Date();
console.log(d.toLocaleString('fr-FR')); // 27/08/2018 à 13:51:11

options = {
  day: 'numeric',
  month: 'long',
  year: 'numeric',
  hour: 'numeric',
  minute: 'numeric',
  timeZoneName: 'long'
};

s = d.toLocaleString('en-US',options)
console.log(s); // August 27, 2018, 2:00 PM Central European Summer Time

s = d.toLocaleString('fr-FR',options)
console.log(s); // 27 août 2018 à 13:57 heure d'été d'Europe centrale
```



RegExp

L'objet RegExp encapsule une **expression régulière** servant à établir la correspondance entre des chaînes de caractères et un motif (*pattern*). [RegExp]

```
p1 = /\w+\.html/g;  
console.log(typeof p1, p1);           // object /\w+\.html/g  
  
p2 = new RegExp('\\w+\\.html', 'g');  
console.log(typeof p2, p2);           // object /\w+\.html/g
```

Il possède des attributs correspondant aux éléments de l'expression littérale :
flags, global, ignoreCase, multiline, source, sticky, unicode

```
p = /\w+.*\.html/g;  
console.log(p.source, p.flags, p.global); // \w+.*\.html g true
```

L'exploitation des expressions régulières se fait via les méthodes *exec* et *test* des objets RegExp, et les méthodes *match*, *search*, *replace* et *split* des objets String.

```
text = document.documentElement.textContent; // texte de ce document  
result = text.match(/\w+\.html/g);  
console.log(result); // Array(3) ["exemple1.html", "exemple2.html", ...]
```

☞ Les méthodes *search*, *replace* et *split* fonctionnent avec des chaînes ou des expressions régulières. Pour des raisons de performances il est préférable d'utiliser si possible des chaînes.

Error

En Javascript l'objet Error possède essentiellement deux attributs (*name* et *message*) correspondant respectivement au nom et au texte de l'erreur : [Error]

```
try {  
  throw new Error('a test');  
}  
catch (e) {  
  console.log(e.name + ': ' + e.message); // Error: a test  
}
```

Pour préciser le type d'une erreur, on peut créer des objets personnalisés, ou utiliser certaines des erreurs natives, qui héritent de celui-ci :

EvalError, InternalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError

```
try {  
  ...  
}  
catch (e) {  
  if ( e instanceof SyntaxError ) {  
    console.log(e.name + ': ' + e.message);  
  }  
  ...  
}
```



2.4.5 Typage canard

JavaScript permet par défaut de modifier dynamiquement un objet existant par ajout de nouveaux attributs, et par modification ou suppression d'attributs existants.

```
c = { espece: "canari", nom: "titi" }; // initialisation
console.log(c); // Object { espece: "canari", nom: "titi" }
c.cri = "tweet"; // ajout
console.log(c); // {espece: "canari", nom: "titi", cri: "tweet"}
c.cri = "cui"; // modification
console.log(c); // { espece: "canari", nom: "titi", cri: "cui" }
delete c.espece; // suppression
console.log(c); // { nom: "titi", cri: "cui" }
```

Ces possibilités s'appliquent même aux objets natifs :

```
a = [1,2,3];
a.nom = 'tableau'; a.espece = "natif"; a.cri = '(?)';
console.log(a); // Array(3) [ 1, 2, 3 ]
console.log(a.nom, a.espece, a.cri); // tableau natif (?)
```

Certains des premiers frameworks Javascript (comme *Prototype*) se sont basés sur ce principe pour étendre les objets natifs avec de nouvelles méthodes. Il est actuellement plutôt considéré qu'il ne s'agit pas d'une bonne pratique. [\[discussion\]](#)

L'expression Duck typing vient de la citation :

« *When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck* »

(Quand je vois un oiseau qui marche comme un canard, nage comme un canard, et cancanne comme un canard, je qualifie cet oiseau de canard)

et fait référence au fait qu'en Javascript c'est l'existence d'une méthode ou d'un attribut qui détermine le comportement d'un objet, et non son appartenance à un type ou une classe particulière.

```
f = x => console.log(
  `${x.nom} est un ${x.espece} qui fait ${x.cri} ${x.cri} !`
); // ES6 template literal

c = { espece: "canari", nom: "titi", cri: "cui" };
m = { espece: "chien", nom: "médor", cri: "ouah" };

a = [1,2,3];
a.nom = 'triplet'; a.espece = "tableau"; a.cri = 'houla';
f(c); // titi est un canari qui fait cui cui !
f(m); // médor est un chien qui fait ouah ouah !
f(a); // triplet est un tableau qui fait houla houla !
```

2.4.6 Objet global

En JavaScript, il existe un objet unique dit l'objet global, qui est créé avant que le contrôle ne passe au programme utilisateur. [\[objet global\]](#)



Toutes les variables (*NaN*, *Infinity*, ...) et les fonctions (*parseInt*, *parseFloat*, *Array*, *String*, ...) natives globales sont des attributs de l'objet global. Dans le cadre d'un navigateur, l'objet global possède un attribut `window` qui se réfère à lui-même :

```
console.log( window === window.window );      // true
console.log( parseInt === window.parseInt );   // true
console.log( Array === window.Array );         // true
```

🔧 Dans un autre contexte qu'un navigateur, l'objet global pourra s'appeler différemment. Sous `node.js` par exemple, l'objet global s'appelle `global`.

Les variables globales déclarées avec `var` sont également des attributs de l'objet global, mais pas celles déclarées avec `let` ou `const`.

```
var a = 3; let b = 4;
console.log( a === window.a );                 // true
console.log( b === window.b );                 // false
```

🔧 A l'époque où Javascript a été conçu pour permettre le développement de scripts dans une page web, l'objet global devait servir à passer des informations (*variables*) entre scripts d'une même page (*window*). Bien que la notion d'espace global soit de moins en moins populaire (*en particulier côté serveur*), l'objet global est resté.

2.4.7 Exercice : utilisation des objets prédéfinis.

Pour les littéraires...

Récupérer le [\[document fourni\]](#), puis le compléter avec du code Javascript pour :

- 1 afficher le nombre de caractères dans l'ensemble des paragraphes,
- 2 même question en ignorant les espaces initiaux et finaux des paragraphes,
- 3 compter le nombre de caractères de ponctuation,
- 4 d'espaces,
- 5 de lettres,
- 6 de consonnes,
- 7 de voyelles.

Références utiles :

[\[Node.textContent\]](#) [\[Document.getElementsByTagName\]](#) [\[spread syntax\]](#) [\[String.prototype.trim\]](#)
[\[String.prototype.match\]](#) [\[Array.prototype.reduce\]](#) [\[RegExp\]](#)



2.5 Fonctions

2.5.1 Fonctions de première classe !

En JavaScript les fonctions sont dites *first-class citizen*, c'est-à-dire qu'elles sont elles-mêmes des objets avec des attributs comme *length* et des méthodes comme *call()*.

```
add = function(a,b) { return a+b; };  
console.log( add.length );           // 2 : nbre d'arguments déclarés  
console.log( add.call(null,2,3) );   // 5 : 2 + 3 = 5
```

Au titre d'objets comme les autres elles peuvent être assignées à des variables, passées en argument lors d'un appel de fonction, retournées par une fonction, et plus généralement manipulées comme tout autre objet.

```
a = [ 18, 5, 3, 10, 1, 8 ];           // tableau à trier  
f = function(a,b) { return(a-b); };   // fonction de comparaison  
a.sort(f);                           // passée en paramètre  
console.log(a);                      // Array(6) [ 1, 3, 5, 8, 10, 18 ]
```

Lorsqu'on dispose d'une référence à une fonction, cette dernière est invoquée à l'aide de l'opérateur *()*.

```
add = function(a,b) { return a + b; }; // ES6 : add = (a,b) => a+b;  
console.log( add(1,2) );               // 3
```

2.5.2 Méthodes

Les méthodes des objets sont simplement des attributs dont la valeur est une fonction.

Au sein d'une méthode, JavaScript affecte l'objet via lequel on appelle la méthode au symbole spécial *this* :

```
parents = {  
  papa : "Raymond",  
  maman : "Ginette",  
  display : function(p) { console.log(p+' : '+this[p]); }  
}  
parents.display("papa"); // papa : Raymond
```

Attention : les fonctions flèche ES6 (*arrow functions*) n'affectent pas le symbole *this*.

```
parents = {  
  papa : "Raymond",  
  maman : "Ginette",  
  display : p => console.log(p+' : '+this[p])  
}  
parents.display("papa"); // papa : undefined
```

2.5.3 Fonctions imbriquées

En JavaScript, les définitions de fonction peuvent être imbriquées.



Une fonction imbriquée a accès aux variables globales et aux variables locales des fonctions parentes.

```
function getPrimes(max) {  
    var n, primes=[1];  
  
    // vérifie si p divise n (renvoie true ou false)  
    function divises (p) { return ( p!=1 && !(n % p) ); }  
  
    // on remplit le tableau des nombres premiers  
    for ( n = 2; n < max; n++ ) {  
        if ( ! primes.some(divises) ) primes.push(n);  
    }  
    return primes;  
}  
console.log(getPrimes(20)); // Array(9) [1, 2, 3, 5, 7, 11, 13, 17, 19]
```

☞ Une variable est d'abord recherchée dans la fonction locale. Si elle n'est pas déclarée dans la fonction locale, elle est recherchée dans la fonction parente, puis ainsi de suite jusqu'au niveau global (*scope chain*).

Dans certains cas (*cf. boucles*) il est utile d'effectuer une copie locale d'une variable pour ne pas multiplier le temps passé à cette recherche... [\[article\]](#)

2.5.4 Clôtures

Les fonctions imbriquées posent un problème intéressant :

```
var papa = "Raymond";  
function famille() {  
    var maman = "Ginette";  
    return function() {  
        console.log("papa : "+papa+", maman : "+maman);  
    };  
}  
f = famille(); // renvoie une fonction...  
f();           // papa : Raymond, maman : Ginette  
papa = "Robert"; // aucun moyen de modifier maman ...  
f();           // papa : Robert, maman : Ginette
```

Ce résultat peut paraître surprenant : lors de l'exécution de la fonction *f()*, la variable *maman* n'existe plus, car la fonction *famille()* a fini de s'exécuter...

Explication : Toute référence (*ici f*) à la fonction (*anonyme*) renvoyée par un appel à *famille*, comporte également les références (*pointeurs*) vers toutes les variables non locales utilisées par cette fonction, qui du coup ne sont pas détruites par le ramasse-miettes. (*garbage collector*) [\[gestion mémoire\]](#)

On appelle cette combinaison (*fonction + référence aux variables non locales*) une fermeture ou une clôture (*closure*). [\[clôtures\]](#)



JavaScript ne propose pas de variables statiques, mais il est facile d'arriver au même résultat à l'aide d'une clôture :

```
incr = function() { // renvoie une fonction
  var value = 0;    // variable "statique"
  return function() { // fonction d'incrément
    return value += 1;
  }
}(); // IIFE

console.log( incr() ); // 1
console.log( incr() ); // 2
```

Cet exemple peut se généraliser pour effectuer d'autres opérations sur la même variable statique :

```
var cpt = function() { // renvoie un objet avec quatre clôtures
  var value = 0;      // variable "statique"
  return {
    inc: function() { ++value; }, // incrément
    dec: function() { --value; }, // décrétement
    reset: function() { value = 0; }, // remise à zéro
    get: function() { return value; } // getter
  }
}(); // IIFE
console.log(cpt.get()); // 0
cpt.dec();
console.log(cpt.get()); // -1
```

[exemple]

2.5.5 Prototypes

Principe des prototypes

Les objets de JavaScript ne sont pas implémentés avec la notion de classe et d'instance, mais utilisent la notion de **prototype**.

Il s'agit d'une forme d'orientation objet dans laquelle le comportement se transmet par clonage d'un objet existant (*le prototype*) et non par héritage.

JavaScript dispose d'un objet de base nommé *Object* dont dérivent tous les autres objets, qui offre un certain nombre de méthodes, et possède un prototype *Object.prototype*.

```
/*
** Le prototype de adam est Object.prototype
** qui est lui-même un objet...
*/
var adam = { genre: "masculin", maison: "Paradis" };
console.log(Object.getPrototypeOf(adam) === Object.prototype); // true

/*
** Le prototype de eve est adam :
** { genre: "masculin", maison: "Paradis" }
*/
var eve = Object.create(adam);
console.log( Object.getPrototypeOf(eve) === adam ); // true
```



Chaîne des prototypes

Lorsqu'on accède à un attribut d'un objet, la valeur renvoyée provient soit de l'objet lui-même (s'il dispose de l'attribut demandé), soit de son prototype.

```
var adam = { genre: "masculin", maison: "Paradis" };
var eve = Object.create(adam); // prototype adam
eve.genre = "féminin";
eve.role = "tentatrice";

var c = console;
c.log(eve);           // ({genre:"féminin", role:"tentatrice"})
c.log(eve.role);      // tentatrice (own property)
c.log(eve.genre);     // féminin (own property)
c.log(eve.maison);    // Paradis (prototype property)
```

☞ Si le prototype possède lui-même un prototype, la recherche d'un attribut se poursuit si nécessaire tout au long de la chaîne des prototypes (*prototype chain*).

Ce mécanisme se rapproche dans son principe de la recherche d'une variable dans l'espace d'une fonction (*scope chain*).

Liste des attributs d'un objet

L'instruction `for ... in` boucle sur **tous** les attributs d'un objet, y compris ceux accessibles via son prototype :

```
var adam = { genre: "masculin", maison: "Paradis" };
var eve = Object.create(adam); // prototype adam
eve.genre = "féminin";
eve.role = "tentatrice";

for ( p in eve ) {
    console.log( p + ": " + eve[p] );
} // genre: féminin, role: tentatrice, maison: Paradis
```

La méthode `Object.prototype.hasOwnProperty()` permet de savoir si un attribut appartient à l'objet lui-même ou provient de la chaîne des prototypes :

```
for ( p in eve ) {
    if ( eve.hasOwnProperty(p) )
        console.log( p + ": " + eve[p] );
} // genre: féminin, role: tentatrice
```

☞ Ce comportement de l'instruction `for ... in` est l'une des raisons pour lesquelles il est déconseillé de modifier les objets natifs.

C'est ainsi que l'utilisation du framework *Prototype* impacte potentiellement les boucles `for ... in` d'un programme qui l'utilise...



Les prototypes sont vivants

La connexion entre un objet et son prototype est une **référence** (*il ne s'agit pas d'une copie*). Toute modification d'un objet impacte donc immédiatement tous les objets dont il est le prototype.

```
var adam = { genre: "masculin", maison: "Paradis" };
var eve = Object.create(adam); // prototype adam
eve.genre = "féminin";
eve.role = "tentatrice";

var c = console;
c.log(eve.maison); // Paradis (attribut du prototype)

adam.maison = "Enfer";
c.log(eve.maison); // Enfer
```

Basé objet vs. orienté objet

N.B. Comme son fonctionnement ne s'appuie pas sur les classes, et ne possède pas explicitement les notions d'héritage voire de polymorphisme, certains qualifient JavaScript de langage "basé objet" et non "Orienté Objet"... [\[object based\]](#).

Ces considérations sont d'autant plus vaines que tous ces comportements peuvent être parfaitement émulés avec les mécanismes mis à disposition par Javascript.

2.5.6 Constructeurs

En JavaScript un constructeur est une fonction appelée avec le mot-clé *new*. Lorsqu'une fonction est appelée de cette manière, le symbole *this* au sein de cette fonction se réfère à un objet *Object* nouvellement créé qui est ensuite implicitement renvoyé par l'appel à *new* :

```
function Humain(nom) {
    this.nom = nom;
}
adam = new Humain('adam');
console.log(adam); // Object { nom: "adam" }
```

N.B. Si la fonction est appelée sans le mot-clé *new*, alors il n'y a pas d'objet créé et *this* se réfère à l'objet global...

Le prototype de l'objet nouvellement créé est initialisé avec la valeur de l'attribut *prototype* du constructeur :

```
console.log(Object.getPrototypeOf(adam) == Humain.prototype); // true

Humain.prototype.maison = "Paradis";
console.log(adam.maison); // Paradis
```

⚠ Attention à ne pas confondre le prototype d'un objet, accessible via la méthode *getPrototypeOf()* implémentée par le prototype d'*Object*, et l'attribut *prototype* des fonctions utilisé pour initialiser le prototype de l'objet créé lorsqu'elles sont appelées avec le mot-clé *new*. [\[détails\]](#)



Par défaut, l'objet correspondant à l'attribut *prototype* d'une fonction comporte un attribut *constructor* initialisé avec la fonction elle-même :

```
function Humain(nom) {  
    this.nom = nom;  
}  
console.log(Humain.prototype.constructor === Humain); // true  
  
eve = new Humain('eve');  
console.log(eve.constructor === Humain); // true;
```

Autres exemples :

```
a = [1, 2, 3, 4];  
console.log(a.constructor === Array); // true  
  
r = /\w+/;  
console.log(r.constructor === RegExp); // true  
  
s = "hello";  
console.log(s.constructor === String); // true
```

... le dernier exemple étant éventuellement surprenant. Pourquoi la valeur de *s* n'est-elle pas du type natif *string* ?

2.5.7 Classes

Si l'orientation objet de Javascript, basée sur les prototypes, est extrêmement puissante et versatile, elle a perturbé plus d'un développeur et notamment les aficionados de Java.

ES6 a ainsi introduit une [nouvelle syntaxe](#) permettant de déclarer des classes de manière plus familière pour de nombreux développeurs :

```
class Rectangle {  
    constructor(height, width) {  
        this.height = height;  
        this.width = width;  
    }  
}  
  
var rect = new Rectangle(100,200);  
console.log(rect.width, rect.height);
```

Il est important de comprendre que cette notation ne remet nullement en compte l'orientation objet basée sur les prototypes, car au-delà de la syntaxe, les classes déclarées de cette manière sont implémentées de manière classique :

```
// Rectangle est une fonction constructeur classique  
console.log(typeof Rectangle); // function  
  
// rect est un objet classique, avec un prototype  
console.log(Object.getPrototypeOf(rect) === Rectangle.prototype); // true
```



Le code de cet exemple :

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
}  
  
var rect = new Rectangle(100,200);  
console.log(rect.width, rect.height);
```

est équivalent à celui-ci :

```
"use strict";  
function _classCallCheck(instance, Constructor) {  
  if (!(instance instanceof Constructor)) {  
    throw new TypeError("Cannot call a class as a function");  
  }  
}  
  
var Rectangle = function Rectangle(height, width) {  
  _classCallCheck(this, Rectangle);  
  
  this.height = height;  
  this.width = width;  
};  
  
var rect = new Rectangle(100, 200);  
console.log(rect.width, rect.height);
```

☞ Ce code est obtenu à l'aide du transcompilateur **Babel**.

Le code transcompilé illustre certaines particularités des déclarations de classe :

- `"use strict";`

Le corps d'une classe est exécuté en mode strict, déjà évoqué par ailleurs

- `var Rectangle = function Rectangle(...){...}`

Contrairement aux déclarations de fonctions, les déclarations de classe ne sont pas hissées (*hoisted*). Une classe ne peut donc pas être utilisée avant d'avoir été déclarée :

```
// ReferenceError: can't access lexical declaration 'Rect' before initialization  
var r = new Rect(10,20);  
class Rect { constructor(h,w) { this.h = h; this.w = w; } }
```

- `_classCallCheck(this, Rectangle);`

Une classe ne peut pas être appelée comme une fonction (*i.e. sans new*) :

```
class Rect { constructor(h,w) { this.h = h; this.w = w; } }  
Rect(10,20); // TypeError: class constructors must be invoked with 'new'
```



Constructeur

La méthode `constructor()` (*mot réservé*) est la méthode permettant de créer une "instance", ou plus précisément (*au sens de javascript*) de créer un objet dont le prototype correspond au modèle défini par la déclaration de classe.

Méthodes du prototype

La définition d'une méthode partagée par toutes les "instances" se fait sans le mot-clé *function* :

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height;  
    this.width = width;  
  }  
  area() {  
    return this.width * this.height;  
  }  
}  
var rect = new Rectangle(10,20);  
console.log(rect.area()); // 200
```

☞ Une méthode partagée par toutes les "instances" est donc un attribut de leur prototype. Pour les plus curieux il peut être judicieux de voir comment le code ci-dessus est transcompilé en ES3 par [Babel](#).

Getters

Associé à la définition d'une méthode, le mot-clé `get` permet de simuler l'accès à un attribut :

```
class Rectangle {  
  constructor(height, width) {  
    this.height = height; this.width = width;  
  }  
  get area() {  
    return this.width * this.height;  
  }  
}  
var rect = new Rectangle(10,20);  
console.log(rect.area); // 200
```

Setters

De même, le mot-clé `set` permet de lier la modification d'un attribut à l'appel d'une fonction :

```
class Circle {  
  constructor(radius) { this.radius = radius; }  
  set diameter(value) { this.radius = value/2; }  
}  
let c = new Circle(10);  
console.log(c.radius); // 10  
c.diameter = 10;  
console.log(c.radius); // 5
```




Méthodes statiques

Certaines méthodes ne sont pas partagées par chacune des instances et peuvent être uniquement invoquées via le constructeur. On parle dans ce cas de méthodes statiques ou de méthodes de classe (*par opposition aux méthodes d'instance*) dans les langages à orientation objet classique.

```
class Rectangle {
  constructor(height, width) {
    this.height = height; this.width = width;
  }
  static definition() {
    return 'Un rectangle est un quadrilatère dont les quatre angles sont droits.';
  }
}
let rect = new Rectangle(10,20);
console.log(Rectangle.definition()); // ok
console.log(rect.definition());      // TypeError: rect.definition is not a function
```

🔧 Le même raisonnement s'applique aux getters et aux setters.

```
class Rectangle {
  ...
  static get definition() {
    return 'Un rectangle est un quadrilatère dont les angles sont droits';
  }
}
console.log(Rectangle.definition());
```

Héritage

```
class Rectangle {
  constructor(w,h) { this.w = w; this.h = h; }
  get area() { return this.w * this.h }
}
class Square extends Rectangle {
  constructor(w) { super(w,w); }
}
let s = new Square(10);
console.log(s.area); // 100
```



Noter l'indication de la classe mère réalisée grâce au mot-clé `extends` et l'appel du constructeur parent à l'aide de la fonction `super()`. `super` peut également être utilisé pour accéder à des méthodes de la classe mère autres que le constructeur :

```
class Rectangle {
  constructor(w,h) { this.w = w; this.h = h; }
  get definition() { return this.constructor.name+' : quadrilatère à angles droi
}
class Carre extends Rectangle {
  constructor(w) { super(w,w); }
  get definition() {
    let def = super.definition;
    return def.substring(0,def.length-1) + ' et côtés de même longueur.'
  }
}
sq = new Carre(10);
console.log(sq.definition); // Carre: quadrilatère à angles droits et côtés de
```

2.5.8 Retour sur this

En JavaScript, le symbole *this* peut prendre différentes valeurs en fonction du contexte d'exécution :

Contexte global

En dehors du corps d'une fonction, la valeur de *this* correspond à l'objet global.

```
console.log( this === window ); // true
```

☞ Dans le contexte d'un module node.js l'objet global est égal à *module.exports*.

Appel d'un constructeur

Au sein d'une fonction appelée avec le mot-clé *new*, *this* correspond au nouvel objet en cours de création :

```
function Humain() { this.self = this; }
adam = new Humain();
console.log( adam.self === adam ); // true
```

Appel d'une méthode

Au sein d'une méthode, la valeur de *this* sera égale à l'objet via lequel la méthode a été appelée :

```
obj = { check: function() { return this; } };
console.log( obj.check() === obj ); // true
```

Appel de fonction

Depuis une fonction appelée de manière standard (*ni méthode ni constructeur*), la valeur de *this* est égale à l'objet global (*soit module.exports sous node.js*) ou vaut *undefined* en mode strict :

```
function f() { return this; }
console.log( f() === window ); // true
```

```
'use strict';
function f() { return this; }
console.log( f() === undefined ); // true
```



Attention, Cette règle est à prendre au pied de la lettre :

```
function Humain(nom) {  
    this.nom = nom;  
  
    function set_type(type) { this.type = type; }  
    set_type('homo sapiens');  
}  
adam = new Humain("adam");  
console.log(adam.nom);           // adam  
console.log(adam.type);          // undefined :-(  
console.log(window.type);        // homo sapiens :-(
```

☞ Vue la forme de l'appel de la fonction **set_type**, **this** correspond à l'objet global...

Cet aspect a perturbé beaucoup de programmeurs. Pour autant, diverses techniques existent pour arriver au but recherché :

```
function Humain(nom) {  
    this.nom = nom;  
  
    var self = this;    // on crée une clôture...  
    function set_type(type) { self.type = type; }  
    set_type('homo sapiens');  
}  
adam = new Humain("adam");  
console.log(adam.type);        // homo sapiens :-)
```

Fonction flèche

Au sein d'une fonction flèche, **this** se comporte comme une variable traditionnelle et sa valeur est égale à celle que possède **this** dans le contexte dans lequel la fonction flèche a été définie.

```
function Humain(nom) {  
    this.nom = nom;  
  
    var set_type = (type) => this.type = type;  
    set_type('homo sapiens');  
}  
adam = new Humain("adam");  
console.log(adam.type);        // homo sapiens :-)
```

☞ Les fonctions flèche ne possèdent pas non plus leurs propres *arguments*...

Affectation explicite

Via la méthode **call()** du prototype de **Function**, il est possible d'affecter explicitement la valeur de **this** lors d'un appel à une fonction :

```
function Humain(nom) {  
    this.nom = nom;  
  
    var set_type = function(type) { this.type = type; };  
    set_type.call(this, 'homo sapiens');  
}  
adam = new Humain("adam");  
console.log(adam.type);        // homo sapiens :-)
```



☞ Le prototype de l'objet Function possède également une méthode *apply()* dont la seule différence est que les arguments sont passés sous forme d'un tableau.

La méthode *bind()* du prototype de Function renvoie une fonction dont *this* est fixé à une valeur passée en argument :

```
function Humain(nom) {  
    this.nom = nom;  
  
    var set_type = function(type) { this.type = type; }.bind(this);  
    set_type('homo sapiens');  
}  
adam = new Humain("adam");  
console.log(adam.type);      // homo sapiens :-)
```

☞ *bind()* permet également de fixer la valeur des premiers arguments. [\[bind\]](#)

Au sujet des gestionnaires d'événements

Un gestionnaire d'événement (*Event Handler*) est une fonction appelée par un programme asynchrone (*callback*) lors de l'occurrence d'un événement.

La plupart des environnements (*APIs, frameworks...*) fonctionnant avec des fonctions de rappel appellent ces dernières en affectant une valeur particulière à *this*.

Un gestionnaire d'événements du DOM (*Document Object Model*) sera par exemple appelé avec *this* correspondant à l'élément qui porte le gestionnaire :

```
<p id="demo_DOMEH">Hello</p>  
<script>  
demo_DOMEH.addEventListener('mouseover', function() {  
    this.style.backgroundColor = '#ccf';  
});  
demo_DOMEH.addEventListener('mouseout', function() {  
    this.style.backgroundColor = 'transparent';  
});  
</script>
```

Conclusion au sujet de *this*

La valeur de *this* au sein d'une fonction dépend uniquement de la façon dont cette fonction a été appelée.

Corollaire : lorsqu'une fonction utilise *this*, éviter de l'appeler d'une manière non prévue...



2.5.9 Exercice : Calcul de moyennes

Le document [\[transiliens-todo.html\]](#) affiche des tableaux avec les statistiques de ponctualité des trains de banlieue parisiens (*RER et transiliens*) mois par mois entre janvier 2013 et août 2015 :

Ponctualité des lignes transiliennes :

Lignes de RER :

RER A :			RER B :			RER C :		
date	ponctualité	satisfaction	date	ponctualité	satisfaction	date	ponctualité	satisfaction
01 - 2013	83.6	5.1	01 - 2013	80.1	4.0	01 - 2013	91.9	11.3
02 - 2013	86.2	6.2	02 - 2013	80.3	4.1	02 - 2013	91.8	11.2
03 - 2013	84.0	5.3	03 - 2013	82.7	4.8	03 - 2013	87.6	7.1
04 - 2013	83.3	5.0	04 - 2013	86.4	6.4	04 - 2013	88.7	7.8

aperçu de transiliens-todo.html

Ce document est outillé via des éléments `` portant les identifiants `ponctualite_globale`, `ponctualite_rer` et `ponctualite_transiliens` dont le contenu est initialement vide.

```
<span id="ponctualite_globale"></span>
```

Ces éléments sont respectivement destinés à recevoir la ponctualité moyenne pour l'ensemble des trains, des RERs, ou des transiliens.

L'entête `<th>` de chacune des tables comporte par ailleurs un élément `` sans attribut `id` destiné à recevoir la valeur moyenne de la ponctualité pour chacune des tables considérées.

```
<th colspan="3" style="...">RER A : <span>&#160;</span></th>
```

Ponctualité des lignes transiliennes : 87.92%

Lignes de RER : 86.38%

RER A : 78.44%			RER B : 85.32%			RER C : 89.38%		
date	ponctualité	satisfaction	date	ponctualité	satisfaction	date	ponctualité	satisfaction
01 - 2013	83.6	5.1	01 - 2013	80.1	4.0	01 - 2013	91.9	11.3
02 - 2013	86.2	6.2	02 - 2013	80.3	4.1	02 - 2013	91.8	11.2
03 - 2013	84.0	5.3	03 - 2013	82.7	4.8	03 - 2013	87.6	7.1
04 - 2013	83.3	5.0	04 - 2013	86.4	6.4	04 - 2013	88.7	7.8

résultat attendu

✎ Compléter ce document avec le code javascript nécessaire pour renseigner les champs destinés à recevoir les valeurs moyennes, qui devront être calculées à partir des valeurs de ponctualité présentes dans le document.

Suggestion : se documenter sur les fonctions `querySelector` [1] [2] et `querySelectorAll` [3] [4] puis créer et exploiter de manière ad'hoc une fonction nommée `ponctualite` retournant la ponctualité moyenne calculée sur l'ensemble des cellules de tableau, cibles d'un sélecteur CSS passé en argument.



2.6 Rapide introduction au DOM

L'interfaçage avec le navigateur s'effectue essentiellement via le DOM (*Document Object Model*).

Le DOM est une API (*Application Programming interface*) ayant donné lieu à plusieurs recommandations du W3C (*World Wide Web Consortium*).

[DOM2 Core] [DOM2 HTML]

Il permet notamment :

- d'accéder aux éléments HTML contenus dans la page, en lecture et en écriture,
- d'accéder au style (CSS) des éléments contenus dans la page, en lecture et en écriture,
- de programmer des gestionnaires d'événement pour réagir aux interactions de l'utilisateur...

2.6.1 Manipulation du DOM

DOM - Récupération d'éléments

`document.getElementById(id)`

Renvoie l'élément possédant l'attribut *id* dont la valeur est spécifiée.

```
<div id="toto">Je suis un paragraphe qui sera pourpre</div>

<script>
  var element_a_colorer = document.getElementById('toto');

  element_a_colorer.style.backgroundColor = 'purple';
  element_a_colorer.style.color = 'white';
</script>
```

Je suis un paragraphe qui sera pourpre

`document.getElementsByClassName(name)`

Renvoie les éléments possédant un attribut *class* avec la valeur spécifiée.

```
<div class="tata">Je suis un paragraphe qui sera bleu</div>
<div>Je suis un paragraphe avec <span class="tata">un mot</span> qui sera bleu</div>

<script>
  var elements_a_colorer = document.getElementsByClassName('tata');

  for ( var n = 0; n < elements_a_colorer.length; n++ ) {
    var element = elements_a_colorer[n];
    element.style.backgroundColor = '#77C';
    element.style.color = 'white';
  }
</script>
```

Je suis un paragraphe qui sera bleu

Je suis un paragraphe avec un mot qui sera bleu

`document.getElementsByTagName(name)`

Renvoie tous les éléments correspondant au nom de balise spécifié.



```
<div id="titi">
  Je suis un paragraphe avec des <span>mots</span>
  qui seront <span>verts</span>
</div>

<script>
  var paragraphe = document.getElementById('titi');

  var elements_a_colorer = paragraphe.getElementsByTagName('SPAN');
  for ( var n = 0; n < elements_a_colorer.length; n++ ) {
    var element = elements_a_colorer[n];
    element.style.backgroundColor = '#4C4';
    element.style.color = 'white';
  }
</script>
```

Je suis un paragraphe avec des mots qui seront verts

`document.querySelector(sel)`

Renvoie le premier élément répondant au sélecteur CSS spécifié.

`document.querySelectorAll(sel)`

Renvoie l'ensemble des éléments répondant au sélecteur CSS spécifié.

```
<div id="tutu">
  Je suis un paragraphe avec des <span>mots</span>
  qui seront <span>roses</span>
</div>

<script>
  var elements_a_colorer = document.querySelectorAll('#tutu span');
  for ( var n = 0; n < elements_a_colorer.length; n++ ) {
    var element = elements_a_colorer[n];
    element.style.backgroundColor = 'pink';
    element.style.color = 'white';
  }
</script>
```

Je suis un paragraphe avec des mots qui seront roses

🔧 Ces deux méthodes sont relativement récentes (02/2013). [\[Selectors API\]](#)

DOM - Création d'éléments

`document.createElement(type)`

Crée un objet Javascript correspondant à un élément du type spécifié (*nom de la balise*), et le renvoie.

`node.appendChild(element)`

Insère l'élément spécifié dans l'arbre des éléments du navigateur, après le dernier enfant du nœud *node*.

```
<div id="riri">Ce texte sera suivi par une ligne horizontale.</div>

<script>
  var paragraphe = document.getElementById('riri');

  var ligne = document.createElement('HR');
  paragraphe.appendChild(ligne);
</script>
```

Ce texte sera suivi par une ligne horizontale.



`document.createTextNode(texte)`

Crée un objet Javascript "noeud texte" et le renvoie.

```
<div id="fifi" style="border: 1px solid #CCC; padding: 5px 10px;">
  Texte initial du paragraphe.<br>
</div>

<script>
  var paragraphe = document.getElementById('fifi');

  var texte = document.createTextNode('Ce contenu a été ajouté par programme. ');
  paragraphe.appendChild(texte);
</script>
```

Texte initial du paragraphe.
Ce contenu a été ajouté par programme.

`node.insertBefore(nouveau,reference)`

Insère un nouveau noeud dans l'arbre des éléments du navigateur. Ce noeud sera un enfant du noeud *node*, placé avant le noeud de référence spécifié.

```
<div id="loulou" style="border: 1px solid #CCC; padding: 5px 10px;">
  <span>Phrase 1.</span><span>Phrase 2.</span><span>Phrase 3.</span></div>

<script>
  var paragraphe = document.getElementById('loulou');
  var phrases = paragraphe.getElementsByTagName('SPAN');

  for ( var n = 1; n < phrases.length; n++ ) {
    var ligne = document.createElement('HR');
    paragraphe.insertBefore(ligne, phrases[n]);
  }
</script>
```

Phrase 1.
Phrase 2.
Phrase 3.

DOM - Autres propriétés et méthodes des noeuds

`node.parentNode`

Correspond au noeud parent du noeud *node*.

`node.childNodes`

Correspond à la liste des noeuds enfants du noeud *node*.

`node.firstChild`

Correspond au premier enfant du noeud *node*.

`node.lastChild`

Correspond au dernier enfant du noeud *node*.

`node.previousSibling`

Correspond au frère précédent du noeud *node*.

`node.nextSibling`

Correspond au frère suivant du noeud *node*.

`node.nodeValue`

Correspond à la valeur textuelle d'un noeud texte.



`node.getAttribute(attr)`

Renvoie la valeur de l'attribut spécifié du noeud *node*.

`node.setAttribute(attr,value)`

Modifie la valeur de l'attribut spécifié du noeud *node*.

Exemple de navigation au sein du DOM

Soient les cellules de la table suivante à colorer de manière appropriée :

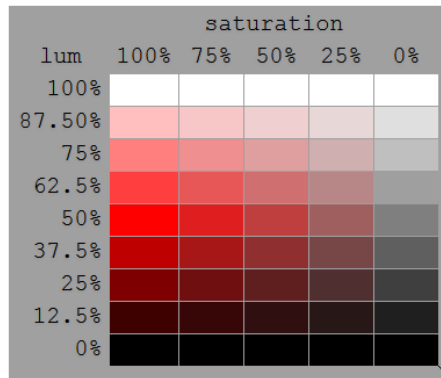
```
<table id="hsl">
  <tr><td> </td><td colspan=5>saturation</td></tr>
  <tr><td>lum</td><td>100%</td><td>75%</td><td>50%</td><td>25%</td><td>0%</td></tr>
  <tr><td>100%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
  <tr><td>87.50%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
  <tr><td>75%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
  <tr><td>62.5%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
  <tr><td>50%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
  <tr><td>37.5%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
  <tr><td>25%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
  <tr><td>12.5%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
  <tr><td>0%</td><td> </td><td> </td><td> </td><td> </td><td> </td></tr>
</table>
```

	saturation				
lum	100%	75%	50%	25%	0%
100%					
87.50%					
75%					
62.5%					
50%					
37.5%					
25%					
12.5%					
0%					

Navigation au sein du DOM

Exemple de solution :

```
<script>                                     // calcul couleur des cellules
var table = document.getElementById('hsl');   // table concernée
var rows = table.getElementsByTagName('TR'); // lignes de la table
for ( var n = 2; n < rows.length; n++ ) {   // boucle sur les lignes
  tr = rows[n];                               // ligne courante
  var lum = tr.firstChild.firstChild.nodeValue; // lum = texte 1ère cellule
  var satNode = rows[1].firstChild.nextSibling; // sat = 2ème cellule 2ème ligne
  for ( var td = tr.childNodes[1]; td; td = td.nextSibling ) {
    var couleur = 'hsl(0,'+satNode.firstChild.nodeValue+', '+lum+')';
    td.style.backgroundColor = couleur;        // couleur de la cellule
    td.setAttribute('title',couleur);         // info-bulle visible au survol
    satNode = satNode.nextSibling;            // sat = cellule suivante ligne 2
  }
}
</script>
```



Style d'un élément

Ainsi qu'aperçu précédemment, le style CSS d'un élément est modifiable via la propriété `style` qui représente la valeur de l'attribut éponyme de cet élément.

Les propriétés de l'objet `style` sont nommées d'après les propriétés CSS en remplaçant les éventuels tirets par une majuscule (*i.e. la propriété CSS `background-color` devient par exemple `style.backgroundColor`*).

```
<p id="dynstyle">Le style de ce paragraphe a été modifié par Javascript</p>

<script>
var paragraphe = document.getElementById('dynstyle');
var style = paragraphe.style;
style.backgroundColor = '#cc8030';
style.color = 'white';
style.padding = '5px 10px';
style.width = '680px';
style.border = '3px outset #cc8030';
</script>
```

Le style de ce paragraphe a été modifié par Javascript

En lecture, l'objet `style` permet d'accéder à la valeur des propriétés CSS d'un élément ayant été modifiées par programme, ou via un attribut `style=`

```
<script>
var style = document.getElementById('dynstyle').style;

var str = 'background-color : '+style.backgroundColor+"\n"+
'color : '+style.color+"\n"+
'padding : '+style.padding+"\n"+
'width : '+style.width+"\n"+
'border : '+style.border+"\n";

document.getElementById('showstyle').appendChild(document.createTextNode(str));
</script>
```

Voici, récupérées grâce au programme ci-dessus, les valeurs des propriétés modifiées du paragraphe page précédente :

```
background-color : rgb(204, 128, 48)
color : white
padding : 5px 10px
width : 680px
border : 3px outset rgb(204, 128, 48)
```



Autres propriétés des éléments HTML

Plus généralement, les objets Javascript correspondant à des éléments HTML possèdent chacun toute une série de propriétés, une pour chacun des attributs autorisés pour l'élément.

```
<p id="otherprops" class="smaller"> Hello </p>

<script>
var p = document.getElementById('otherprops');
p.align = 'center';
p.title = "Le contenu de ce paragraphe a été centré";

var str = 'p.id = '+p.id+"\n"+'p.class = '+p.className;

document.getElementById('showprops').appendChild(document.createTextNode(str));
</script>
```

```
p.id = otherprops
p.class = smaller
```

2.6.2 Gestion des événements

Gestionnaires d'événements HTML4

En HTML 4.0 les gestionnaires d'événements sont spécifiés sous la forme d'attributs. De ce fait il ne peut y avoir qu'un seul gestionnaire par événement :

```
<button onclick="modifier_paragraphe()">Hello</button>

<script>
function modifier_paragraphe() {
  var p = document.getElementById('p1');
  p.style.backgroundColor = 'tomato';
  p.style.color = 'white';
  p.style.padding = '5px 10px';
  p.replaceChild(document.createTextNode('Paragraphe modifié !'),p.firstChild);
}
</script>

<p id="p1">Ce paragraphe sera modifié en actionnant le bouton.</p>
```

On y va !

Ce paragraphe sera modifié en actionnant le bouton.

On y va !

Paragraphe modifié !



Gestionnaires d'événements HTML4 via Javascript

Comme pour les autres attributs, les gestionnaires d'événements HTML 4.0 peuvent être accédés directement sous forme d'une propriété de l'élément concerné :

```
<button id="button2"> OK </button>

<script>
document.getElementById('button2').onclick = function() {
    var p = document.getElementById('p2');
    p.style.backgroundColor = 'darkslateblue';
    p.style.color = 'white';
    p.style.padding = '5px 10px';
    p.replaceChild(document.createTextNode('Paragraphe modifié !'),p.firstChild);
}
</script>

<p id="p2">Ce paragraphe sera modifié en actionnant le bouton.</p>
```

OK

Ce paragraphe sera modifié en actionnant le bouton.

OK

Paragraphe modifié !

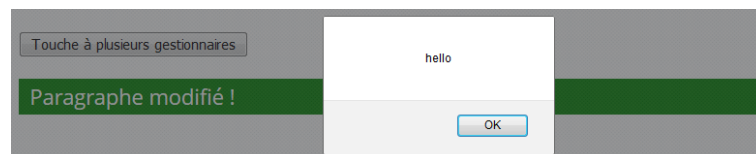
Gestionnaires d'événements : modèle du DOM

Le modèle de gestion des événements du DOM permet l'enregistrement de plusieurs gestionnaires pour le même événement d'un élément donné : [\[DOM2 Events\]](#)

```
<button id="multibutton"> Touche à plusieurs gestionnaires </button>

<script>
function modifier_para() {
    var p = document.getElementById('p3');
    p.style.backgroundColor = 'green';
    p.style.color = 'white';
    p.style.padding = '5px 10px';
    p.replaceChild(document.createTextNode('Paragraphe modifié !'),p.firstChild);
}
var button = document.getElementById('multibutton');
button.addEventListener('click', modifier_para );
button.addEventListener('click', function(){ alert('hello'); } );
</script>

<p id="p3">Ce paragraphe sera modifié en actionnant le bouton.</p>
```



HTML Events

Les événements suivants sont regroupés sous l'appellation "HTML Events" :



événement	éléments source	déclenchement
load	window, frameset, object	au chargement de la page
unload	body, frameset	au passage à une autre page
select	input, textarea	sur sélection de texte
change	input, select, textarea	si contenu modifié
submit	form	lors de la soumission
reset	form	sur réinitialisation
focus	label,input, select, textarea, button	sur prise du focus
blur	label,input, select, textarea, button	sur perte du focus
resize	window	lors d'un modification de taille
scroll	window	sur déroulement de la fenêtre

HTML events

```
var sc = 0;
window.onscroll = function() {
    var pre = document.getElementById('scrolled');
    pre.replaceChild(document.createTextNode('scrolled '+(sc++)), pre.firstChild);
}
```

scrolled 37

Mouse Events

Les événements de la catégorie Mouse Events sont :

événement	éléments source	déclenchement
click	tous éléments	lors d'un clic de souris (down-up-click)
mousedown	tous éléments	au début d'un clic de souris
mouseup	tous éléments	à la fin d'un clic de souris
mouseover	tous éléments	lors d'un survol du curseur
mousemove	tous éléments	lors d'un déplacement du curseur
mouseout	tous éléments	lorsque le curseur quitte l'élément

Mouse Events

```
var p = document.getElementById('mouse-style');
p.addEventListener('mouseover', function(){
    this.style.backgroundColor = '#cc8030';
    this.style.color = 'white';
    this.style.border = '3px outset #cc8030';
});
```

Le style de ce paragraphe est modifié en fonction de la position du curseur

Le style de ce paragraphe est modifié en fonction de la position du curseur

Le style de ce paragraphe est modifié en fonction de la position du curseur

Question : quels sont les autres gestionnaires d'événement nécessaires pour obtenir un comportement correct lors d'un clic (bouton enfoncé) et que le bouton retrouve son aspect initial lorsque le curseur le quitte.



Détails de l'événement

À l'occurrence d'un événement, le gestionnaire est appelé avec un paramètre correspondant à un objet du type `MouseEvent`. Cet objet possède entre autres les propriétés suivantes :

```
long      detail;           // détail - cf. cas par cas
long      screenX;          // abscisse du clic (coords écran)
long      screenY;          // ordonnée du clic (coords écran)
long      clientX;           // abscisse du clic
long      clientY;           // ordonnée du clic
boolean   ctrlKey;           // touche 'ctrl' actionnée
boolean   shiftKey;          // touche 'shift' actionnée
boolean   altKey;            // touche 'alt' actionnée
boolean   metaKey;           // touche 'meta' actionnée
unsigned short button;       // bouton souris actionné (0,1,2)
EventTarget currentTarget;   // élément porteur du gestionnaire
EventTarget relatedTarget;   // élément - cf. cas pas cas
```

```
var mp = document.getElementById('multicolore');
mp.parentNode.addEventListener('mousemove', function(event){
    var h = 180 + Math.atan2(event.clientY-450,event.clientX-550)*180/Math.PI;
    mp.style.backgroundColor = mp.style.borderColor = 'hsl('+h+',100%,50%)';
});
```

La couleur de ce paragraphe varie en fonction de la position du curseur...



2.6.3 Exercices

Le document [\[hsl_todo.html\]](#) affiche un tableau destiné à illustrer le principe de la notation hsl pour exprimer une couleur en CSS :

```
color: hsl(0,100%,50%);
```

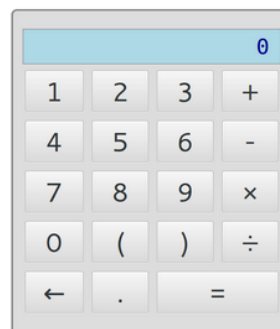
	saturation				
lum	100%	75%	50%	25%	0%
100%					
87.50%					
75%					
62.5%					
50%					
37.5%					
25%					
12.5%					
0%					

	saturation				
lum	100%	75%	50%	25%	0%
100%					
87.50%					
75%					
62.5%					
50%					
37.5%					
25%					
12.5%					
0%					

tableau hsl à compléter (à g.) et pour hue = 0° (à d.)

- 1 À partir du document fourni, remplir les cellules de ce tableau avec les couleurs appropriées, en fonction des valeurs extraites des entêtes de ligne (*pour la luminance*) et de colonne (*pour la saturation*) et une valeur de *hue* arbitraire, par exemple comme ici 0°.
- 2 Faire varier *h* avec la position de la souris.
- 3 Représenter un curseur que l'on peut déplacer avec la souris, permettant de régler *h* entre 0 et 360°.
- 4 Créer un document html représentant une calculatrice, que l'on rendra fonctionnelle à l'aide de javascript.

Bien sûr, tout est possible : calculatrice avec les 4 opérations de base, avec ou sans mémoire, ou avec des fonctionnalités avancées (*fonctions mathématiques, statistiques, conversions de base...*)



exemple de calculatrice



trueRéférences et liens

[Javascript for Acrobat], p.4

ADOBE, "*Javascript for Acrobat*", 2014.

En ligne, disponible sur <http://www.adobe.com/devnet/acrobat/javascript.html>

[consulté le 6 fév. 2024]

[Games by Brent Silby], p.4

BRENT SILBY, "*DEF-LOGIC Games by Brent Silby*", 2001 - 2016.

En ligne, disponible sur <http://def-logic.com/>

[consulté le 6 fév. 2024]

[Instant adventure gaming], p.4

MARTIN KOOL, "*Sarien.net - Instant adventure gaming*", 2011.

En ligne, disponible sur <http://sarien.net/>

[consulté le 6 fév. 2024]

[ActionScript], p.4

"*ActionScript*", Wikipédia, 2 avril 2018.

En ligne, disponible sur <https://fr.wikipedia.org/wiki/ActionScript>

[consulté le 6 fév. 2024]

[V8 engine], p.4

"*V8 (moteur JavaScript)*", Wikipédia, 20 juin 2018.

En ligne, disponible sur https://fr.wikipedia.org/wiki/V8_%28moteur_JavaScript%29

[consulté le 6 fév. 2024]

[Node.js], p.4

"*Node.js*", Wikipédia, 23 juil. 2018.

En ligne, disponible sur <https://fr.wikipedia.org/wiki/Node.js>

[consulté le 6 fév. 2024]

[Node.js], p.4

NODE.JS FOUNDATION, "*Node.js – a Javascript runtime built on Chrome's V8 Javascript engine*", Joyent, Inc., 2018.

En ligne, disponible sur <http://nodejs.org/>

[consulté le 6 fév. 2024]

[Angular], p.4

"*Angular – One framework, Mobile & desktop*", Google, 2010 - 2018.

En ligne, disponible sur <https://angular.io/>

[consulté le 6 fév. 2024]

[Atom], p.4

"*A hackable text editor for the 21st Century*", Github.

En ligne, disponible sur <https://atom-editor.cc/>

[consulté le 6 fév. 2024]

[Express], p.4

STRONGLOOP, IBM et al., "*Express – Fast, unopiniated, minimalist web framework for Node.js*", Node.js Foundation, 2017.

En ligne, disponible sur <https://expressjs.com/>

[consulté le 6 fév. 2024]



[Ionic], p.4

DRIFTY, *"Ionic – Build amazing apps in one codebase, for any platform, with the web."*, Ionic, 2018.

En ligne, disponible sur <https://ionicframework.com/>
[consulté le 6 fév. 2024]

[Meteor], p.4

"Meteor – The fasted way to build Javascript apps.", Meteor Development Group Inc., 2018.

En ligne, disponible sur <https://www.meteor.com/>
[consulté le 6 fév. 2024]

[stackoverflow languages], p.4

"Stackoverflow 2023 Developer Survey - Programming, scripting, and markup languages ", stackoverflow, 2023.

En ligne, disponible sur <https://survey.stackoverflow.co/2023/#section-most-popular-technologies-programming-scripting-and-markup-languages>
[consulté le 6 fév. 2024]

[stackoverflow frameworks], p.4

"Stackoverflow 2023 Developer Survey - Web frameworks and technologies ", stackoverflow, 2023.

En ligne, disponible sur <https://survey.stackoverflow.co/2023/#section-most-popular-technologies-web-frameworks-and-technologies>
[consulté le 6 fév. 2024]

[github], p.5

"Octoverse: The state of open source and rise of AI in 2023 - The most popular programming languages", GitHub Inc., © 2018.

En ligne, disponible sur <https://github.blog/2023-11-08-the-state-of-open-source-and-ai/#the-most-popular-programming-languages>
[consulté le 6 fév. 2024]

[Les langages les plus utilisés en 2023], p.5

LIONEL SUJAY VAILSHERY, *"Most widely utilized programming languages among developers worldwide 2023 "*, statista.com, 19 Jan. 2024.

En ligne, disponible sur <https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/>
[consulté le 6 fév. 2024]

[must read], p.5

S. PEYROTT, *"A Brief History of JavaScript"*, Auth0 Inc., 16 jan. 2017.

En ligne, disponible sur <https://auth0.com/blog/a-brief-history-of-javascript/>
[consulté le 6 fév. 2024]

[archive], p.5

NETSCAPE & SUN, *"Netscape and Sun announce Javascript, the open, cross-platform object scripting language for enterprise networks and the internet"*, Déc. 1995.

En ligne, disponible sur <https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html>
[consulté le 6 fév. 2024]

[JScript vs. ECMA], p.5

MICROSOFT, *"Microsoft JScript, fonctionnalités ECMA"*, 2014.

En ligne, disponible sur <http://msdn.microsoft.com/fr-fr/library/49zhkzs5%28v=vs.100%29.aspx>
[consulté le 6 fév. 2024]



[ECMAScript 1], p.5

"*ECMAScript: A general purpose, cross-platform programming language*", ECMA International, Jun. 1997.

En ligne, disponible sur https://ecma-international.org/wp-content/uploads/ECMA-262_1st_edition_june_1997.pdf
[consulté le 6 fév. 2024]

[ECMAScript 3], p.6

"*ECMAScript Language Specification*", ECMA International, Déc. 1999.

En ligne, disponible sur https://www.ecma-international.org/wp-content/uploads/ECMA-262_3rd_edition_december_1999.pdf
[consulté le 6 fév. 2024]

[article], p.6

D. CROCKFORD, "*JavaScript: The World's Most Misunderstood Programming Language*", 2001.

En ligne, disponible sur <http://crockford.com/javascript/javascript.html>
[consulté le 6 fév. 2024]

[AJAX], p.6

J.J. GARRETT, "*Ajax: A New Approach to Web Applications*", Adaptive Path, 18 fév. 2005.

En ligne, disponible sur <https://web.archive.org/web/20190405130752/http://adaptivepath.org/ideas/ajax-new-approach-web-applications/>
[consulté le 6 fév. 2024]

[XMLHttpRequest], p.6

E. SHEPHERD *et al.*, "*XMLHttpRequest*", Mozilla Developer Network, 31 Juil. 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>
[consulté le 6 fév. 2024]

[Prototype], p.6

"*Prototype, A foundation for ambitious web user interfaces*", Prototype Core Team.

En ligne, disponible sur <http://prototypejs.org/>
[consulté le 6 fév. 2024]

[Dojo], p.6

"*Dojo Toolkit 1.14*", JS Foundation.

En ligne, disponible sur <http://dojotoolkit.org/>
[consulté le 6 fév. 2024]

[jQuery], p.6

"*jQuery, write less, do more*", The jQuery Foundation.

En ligne, disponible sur <http://jquery.com/>
[consulté le 6 fév. 2024]

[ES4 story], p.6

S. PEYROTT, "*The Real Story Behind ECMAScript 4*", Auth0 Inc., 1 mars. 2017.

En ligne, disponible sur <https://auth0.com/blog/the-real-story-behind-es4/>
[consulté le 6 fév. 2024]

[ECMAScript 5], p.6

"*ECMAScript Language Specification - 5th edition*", ECMA International, Déc. 2009.

En ligne, disponible sur https://www.ecma-international.org/wp-content/uploads/ECMA-262_5th_edition_december_2009.pdf
[consulté le 6 fév. 2024]



[fonctionnalités ES6], p.6

S. PEYROTT, "*A Rundown of JavaScript 2015 features*", Auth0 Inc., 16 nov. 2016.
En ligne, disponible sur <https://auth0.com/blog/a-rundown-of-es6-features/>
[consulté le 6 fév. 2024]

[compatibilité ES6], p.6

J. ZAYTSEV, LEON, D. PUSHKAREV, "*ECMAScript 6 compatibility table*", 11 août 2018.
En ligne, disponible sur <https://web.archive.org/web/20230603205024/http://kangax.github.io/compat-table/es6/>
[consulté le 6 fév. 2024]

[Babel], p.6

D. TSCHINDER, *et al.*, "*Babel is a JavaScript compiler. Use next generation JavaScript, today.*", 2018.
En ligne, disponible sur <https://babeljs.io/>
[consulté le 6 fév. 2024]

[notepad++], p.7

DON HO, "*Notepad++*", .
En ligne, disponible sur <http://notepad-plus-plus.org/>
[consulté le 6 fév. 2024]

[Vim], p.7

BRAM MOOLENAAR *et al.*, "*Vim the editor*", .
En ligne, disponible sur <http://www.vim.org/>
[consulté le 6 fév. 2024]

[Sublime Text], p.7

"*Sublime Text*", Sublime HQ Pty Ltd.
En ligne, disponible sur <http://www.sublimetext.com/>
[consulté le 6 fév. 2024]

[Atom], p.7

"*A hackable text editor for the 21st Century*", Github.
En ligne, disponible sur <https://atom-editor.cc/>
[consulté le 6 fév. 2024]

[9 Best JS IDEs], p.7

"*9 Best JavaScript IDE & Source Code Editors [2024]*", InterviewBit, 8 jan. 2024.
En ligne, disponible sur <https://www.interviewbit.com/blog/javascript-ide/>
[consulté le 6 fév. 2024]

[Webstorm], p.7

"*WebStorm – The smartest JavaScript IDE*", Jet Brains.
En ligne, disponible sur <https://www.jetbrains.com/webstorm/>
[consulté le 6 fév. 2024]

[Visual Studio], p.7

"*Visual Studio Code*", Microsoft.
En ligne, disponible sur <https://code.visualstudio.com/>
[consulté le 6 fév. 2024]

[Licence étudiant], p.7

"*Free individual licenses for students and faculty members*", Jet Brains.
En ligne, disponible sur <https://www.jetbrains.com/student/>
[consulté le 4 fév. 2020]



[Can I Use], p.7

A. DEVERIA, "*Can I use ... ?*", caniuse.com.
En ligne, disponible sur <http://caniuse.com>
[consulté le 6 fév. 2024]

[Firefox], p.7

"*Le nouveau Firefox – Toujours plus rapide.*", Mozilla.
En ligne, disponible sur <http://www.mozilla.org/fr/firefox/new/>
[consulté le 6 fév. 2024]

[Chrome], p.7

"*Chrome - Naviguez en toute rapidité*", Google.
En ligne, disponible sur <https://www.google.fr/intl/fr/chrome/browser/>
[consulté le 6 fév. 2024]

[Opera], p.7

"*Opera – Le navigateur rapide, sécurisé et intuitif*", Opera Software ASA.
En ligne, disponible sur <http://www.opera.com/fr>
[consulté le 6 fév. 2024]

[Safari], p.7

"*Safari - La toile a trouvé son maître*", Apple Inc..
En ligne, disponible sur <http://www.apple.com/fr/safari/>
[consulté le 6 fév. 2024]

[Edge], p.7

"*Microsoft Edge – The faster, safer way to get things done on the web.*", Microsoft.
En ligne, disponible sur <https://www.microsoft.com/en-us/windows/microsoft-edge>
[consulté le 6 fév. 2024]

[exemple 0], p.9

D. MULLER, "*Document HTML pour programme Javascript*", 23 jan. 2014, 1 p.
En ligne, disponible sur <http://dmolinarius.github.io/demofiles/elc-d3/cours1/exemple0.html>
[consulté le 21 août 2018]

[consulter], p.9

BRIAN KERNIGHAN, DENNIS RITCHIE, "*The C Programming Language*", Prentice Hall, 1978, p. 5-6.
En ligne, disponible sur http://www.amazon.com/C-Programming-Language-2nd-Edition/dp/0131103628#reader_0131103628
[consulté le 6 fév. 2024]

[exemple 1], p.10

D. MULLER, "*Afficher un message avec document.write*", 23 jan. 2014, 1 p.
En ligne, disponible sur <http://dmolinarius.github.io/demofiles/elc-d3/cours1/exemple1.html>
[consulté le 21 août 2018]

[exemple 2], p.10

D. MULLER, "*Afficher un message avec document.write*", 23 jan. 2014, 1 p.
En ligne, disponible sur <http://dmolinarius.github.io/demofiles/elc-d3/cours1/exemple2.html>
[consulté le 21 août 2018]

[exemple 3], p.11

D. MULLER, "*Afficher un popup avec alert*", 23 jan. 2014, 1 p.
En ligne, disponible sur <http://dmolinarius.github.io/demofiles/elc-d3/cours1/exemple3.html>
[consulté le 21 août 2018]



[exemple 4], p.11

D. MULLER, "*Afficher un message dans la console*", 23 jan. 2014, 1 p.

En ligne, disponible sur <http://dmolinarius.github.io/demofiles/elc-d3/cours1/exemple4.html>

[consulté le 21 août 2018]

[API console], p.12

MDN CONTRIBUTORS, "*The Console Object*", Mozilla Developer Network, 21 déc. 2021.

En ligne, disponible sur <https://developer.mozilla.org/docs/Web/API/console>

[consulté le 6 fév. 2024]

[au choix], p.13

M. CLEMENT, "*JavaScript Semicolon Insertion - Everything you need to know*", 28 mai 2010.

En ligne, disponible sur http://inimino.org/~inimino/blog/javascript_semicolons

[consulté le 6 fév. 2024]

[optionnel], p.13

M. MAROHNIC, "*Semicolons in JavaScript are optional*", 07 Mai 2010.

En ligne, disponible sur <https://mislav.net/2010/05/semicolons/>

[consulté le 6 fév. 2024]

[ASI], p.13

MDN CONTRIBUTORS, "*Automatic semicolon insertion*", Mozilla Developer Network, 8 nov. 2023.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Automatic_semicolon_insertion

[consulté le 6 fév. 2024]

[déclaration de fonction], p.14

BEYTEK *et al.*, "*function declaration*", Mozilla Developer Network, 27 mai 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function>

[consulté le 21 août 2018]

[expression de fonction], p.14

L. ANDREI *et al.*, "*function expression*", Mozilla Developer Network, 29 juin 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function>

[consulté le 21 août 2018]

[fonctions flèche], p.14

H. SHARMA *et al.*, "*Arrow functions*", Mozilla Developer Network, 17 août 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

[consulté le 21 août 2018]

[article 2012], p.16

K. PI, "*10 design flaws of JavaScript*", PixelsTech, 29 nov. 2012.

En ligne, disponible sur <https://www.pixelstech.net/article/1354210754-10-design-flaws-of-JavaScript>

[consulté le 22 août 2018]

[Strict mode], p.16

B. MC CORMICK *et al.*, "*Strict mode*", Mozilla Developer Network, 24 juin 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

[consulté le 22 août 2018]



[ESLint], p.16

"*ESLint - The pluggable linting utility for JavaScript and JSX*", JS Foundation, 17 août 2018.

En ligne, disponible sur <https://eslint.org/>

[consulté le 22 août 2018]

[JSHint], p.16

R. WALDRON *et al.*, "*JSHint, A Static Code Analysis Tool for JavaScript*", 5 août 2018.

En ligne, disponible sur <http://jshint.com/>

[consulté le 22 août 2018]

[TSLint], p.16

"*TSLint - An extensible linter for the TypeScript language.*", Palantir Technologies, 17 nov. 2016.

En ligne, disponible sur <https://palantir.github.io/tslint/>

[consulté le 22 août 2018]

[hoisting], p.16

I. GERCHEV, "*Demystifying JavaScript Variable Scope and Hoisting*", SitePoint Pty. Ltd., 2 déc. 2014.

En ligne, disponible sur <https://www.sitepoint.com/demystifying-javascript-variable-scope-hoisting/>

[consulté le 22 août 2018]

[let], p.16

BERGUS *et al.*, "*let*", Mozilla Developer Network, 15 mai 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>

[consulté le 22 août 2018]

[IIFE], p.17

C. MILLS *et al.*, "*IIFE - Immediately Invoked Function Expression*", Mozilla Developer Network, 18 juil. 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Glossary/IIFE>

[consulté le 22 août 2018]

[const], p.17

M. FUJIMOTO *et al.*, "*const*", Mozilla Developer Network, 16 août 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const>

[consulté le 22 août 2018]

[types de données], p.17

PLUG-N-PLAY *et al.*, "*JavaScript data types and data structures*", Mozilla Developer Network, 19 août 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

[consulté le 22 août 2018]

[IEEE 754], p.18

D. MULLER, "*Le standard IEEE 754*", 18 mars 1998.

En ligne, disponible sur http://dmolinarius.github.io/demofiles/microp/numeration/flp_ieee.html

[consulté le 22 août 2018]

[JS et Unicode], p.19

D. PAVLUTIN, "*What every JavaScript developer should know about Unicode*", 14 Sep. 2016.

En ligne, disponible sur <https://dmitripavlutin.com/what-every-javascript-developer-should-know-about-unicode/>

[consulté le 23 août 2018]



[Symbol], p.19

J. SPHINX *et al.*, "Symbol", Mozilla Developer Network, 15 fév. 2018.
En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol
[consulté le 23 août 2018]

[TypeScript], p.20

"TypeScript – JavaScript that scales", Microsoft, 23 août 2018.
En ligne, disponible sur <http://www.typescriptlang.org/>
[consulté le 23 août 2018]

[Angular], p.20

"Angular – One framework. Mobile & desktop", Google, 2018.
En ligne, disponible sur <https://angular.io/>
[consulté le 23 août 2018]

[réf. Wikipédia], p.22

"Suite de Fibonacci", Wikipédia, 24 déc. 2018.
En ligne, disponible sur https://fr.wikipedia.org/wiki/Suite_de_Fibonacci
[consulté le 08 jan. 2019]

[réf. Wikipédia], p.22

"Nombre premier", Wikipédia, 17 déc. 2018.
En ligne, disponible sur https://fr.wikipedia.org/wiki/Nombre_premier
[consulté le 08 jan. 2019]

[réf. en ligne], p.22

BORIS GOURÉVITCH, "L'univers de π ", 13 avril 2013.
En ligne, disponible sur <http://www.pi314.net/fr/leibniz.php>
[consulté le 08 jan. 2019]

[réf. en ligne], p.22

GÉRARD VILLEMEN, "Constante Pi (π)", 17 oct. 2017.
En ligne, disponible sur <http://villemien.gerard.free.fr/Wwwgvm/Geometri/PiDebut.htm#formule>
[consulté le 08 jan. 2019]

[objets natifs], p.24

F. SCHOLZ *et al.*, "Standard built-in objects", Mozilla Developer Network, 11 août 2018.
En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects
[consulté le 24 août 2018]

[APIs Web], p.24

S. BLUMBERG *et al.*, "Web APIs", Mozilla Developer Network, 13 juil. 2018.
En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/API>
[consulté le 24 août 2018]

[API node.js], p.24

VSE MOZHET BYT *et al.*, "Node.js v10.9.0 Documentation", Node.js Foundation, 14 juil. 2018.
En ligne, disponible sur <https://nodejs.org/api/index.html>
[consulté le 24 août 2018]

[Boolean], p.25

J. SPHINX *et al.*, "Boolean", Mozilla Developer Network, 11 mai 2018.
En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Boolean
[consulté le 24 août 2018]



[Number], p.25

F. SCHOLZ *et al.*, "*Boolean*", Mozilla Developer Network, 8 août 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number

[consulté le 24 août 2018]

[Math], p.26

M. SEYRAWAN *et al.*, "*Math*", Mozilla Developer Network, 14 juil. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

[consulté le 24 août 2018]

[String], p.27

J. GIFFIN *et al.*, "*String*", Mozilla Developer Network, 15 juil. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

[consulté le 24 août 2018]

[JSON], p.27

F. SCHOLZ *et al.*, "*JSON*", Mozilla Developer Network, 4 juil. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON

[consulté le 26 août 2018]

[Array], p.28

XUXINTAO *et al.*, "*Array*", Mozilla Developer Network, 20 juin 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

[consulté le 25 août 2018]

[Date], p.29

P. WANG *et al.*, "*Date*", Mozilla Developer Network, 7 août 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date

[consulté le 27 août 2018]

[expression régulière], p.30

M. GORGINFAR *et al.*, "*Regular Expressions*", Mozilla Developer Network, 28 juil. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

[consulté le 27 août 2018]

[RegExp], p.30

J. STELOVSKY *et al.*, "*RegExp*", Mozilla Developer Network, 30 juil. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp

[consulté le 27 août 2018]

[Error], p.30

KDEX *et al.*, "*Error*", Mozilla Developer Network, 25 avril 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

[consulté le 27 août 2018]

[Prototype], p.31

"*Prototype, A foundation for ambitious web user interfaces*", Prototype Core Team.

En ligne, disponible sur <http://prototypejs.org/>

[consulté le 27 août 2018]



[discussion], p.31

J. AN BUSCHTÖNS, Stack Overflow, 25 déc. 2012.

En ligne, disponible sur <https://stackoverflow.com/questions/14034180/why-is-extending-native-objects-a-bad-practice>

[objet global], p.31

I. KANTOR *et al.*, "Global object", Javascript.info, 2007-2018.

En ligne, disponible sur <https://javascript.info/global-object>
[consulté le 28 août 2018]

[document fourni], p.32

D. MULLER, "Le compte est bon !", 04 déc. 2018, 1 p.

En ligne, disponible sur http://dmolinarius.github.io/demofiles/elc-d3/cours1/wordcount_todo.html
[consulté le 8 jan. 2019]

[Node.textContent], p.32

MARK FLUEHR *et al.*, "Node.textContent", Mozilla Developer Network, 7 déc. 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/API/Node/textContent>
[consulté le 09 jan. 2019]

[Document.getElementsByTagName], p.32

MASAHIRO FUJIMOTO *et al.*, "Document.getElementsByTagName", Mozilla Developer Network, 6 déc. 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementsByTagName>
[consulté le 09 jan. 2019]

[spread syntax], p.32

KADIR TOPAL *et al.*, "spread syntax", Mozilla Developer Network, 13 nov. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
[consulté le 09 jan. 2019]

[String.prototype.trim], p.32

ZOLTÁN BEDI *et al.*, "String.prototype.trim", Mozilla Developer Network, 14 sept. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/Trim
[consulté le 09 jan. 2019]

[String.prototype.match], p.32

WILL BAMBERG *et al.*, "String.prototype.match", Mozilla Developer Network, 10 août 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match
[consulté le 09 jan. 2019]

[Array.prototype.reduce], p.32

VITALII BURLAI *et al.*, "Array.prototype.reduce", Mozilla Developer Network, 1er déc. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/reduce
[consulté le 01 jan. 2019]

[RegExp], p.32

ROBERT MUNRO *et al.*, "RegExp", Mozilla Developer Network, 15 nov. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp
[consulté le 09 jan. 2019]



[[article](#)], p.34

D. CASTORINA, "*JavaScript Prototype Chains, Scope Chains, and Performance: What You Need to Know*", Toptal.

En ligne, disponible sur <https://www.toptal.com/javascript/javascript-prototypes-scopes-and-performance-what-you-need-to-know>

[consulté le 28 août 2018]

[[gestion mémoire](#)], p.34

C. KLIPPEL *et al.*, "*Memory Management*", Mozilla Developer Network, 24 juin 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management

[consulté le 28 août 2018]

[[clôtures](#)], p.34

L. BELSKY *et al.*, "*Closures*", Mozilla Developer Network, 28 mai 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures>

[consulté le 28 août 2018]

[[exemple](#)], p.35

D. MULLER, "*Clôtures Javascript*", 29 août 2018, 1 p.

En ligne, disponible sur <http://dmolinarius.github.io/demofiles/elc-d3/cours1/closures.html>

[consulté le 29 août 2018]

[[object based](#)], p.37

TOM.REDING *et al.*, "*Object-based language*", Wikipédia, 21 oct. 2023.

En ligne, disponible sur https://en.wikipedia.org/wiki/Object-based_language

[consulté le 6 fév. 2024]

[[détails](#)], p.37

J. SPHINX *et al.*, "*prototype and Object.getPrototypeOf*", Mozilla Developer Network, 29 juil. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#prototype_and_Object.getPrototypeOf

[consulté le 28 août 2018]

[[nouvelle syntaxe](#)], p.38

NEIL KAKKAR *et al.*, "*Classes*", Mozilla Developer Network, 6 nov. 2018.

En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes>

[consulté le 9 jan. 2019]

[[Babel](#)], p.39

THE BABEL TEAM, "*Babel - Try it out !*", .

En ligne, disponible sur <https://babeljs.io/repl>

[consulté le 09 jan. 2019]

[[bind](#)], p.44

H. SRIVASTAVA *et al.*, "*Function.prototype.bind()*", Mozilla Developer Network, 26 août. 2018.

En ligne, disponible sur https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

[consulté le 29 août 2018]

[[transiliens-todo.html](#)], p.45

D. MULLER, "*Ponctualité des lignes transiliennes*", 04 déc. 2018, 1 p.

En ligne, disponible sur http://dmolinarius.github.io/demofiles/elc-d3/cours1/moyenne-transiliens_todo.html

[consulté le 7 fév. 2019]



[1], p.45

Y. DAWADI *et al.*, "*Document.querySelector()*", Mozilla Developer Network, 27 janvier 2019.
En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector>
[consulté le 7 fév. 2019]

[2], p.45

M. FLUEHR *et al.*, "*Element.querySelector()*", Mozilla Developer Network, 31 déc. 2018.
En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/API/Element/querySelector>
[consulté le 7 fév. 2019]

[3], p.45

M. FLUEHR *et al.*, "*Document.querySelectorAll()*", Mozilla Developer Network, 4 fév. 2019.
En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelectorAll>
[consulté le 7 fév. 2019]

[4], p.45

J. FINLINSON *et al.*, "*Element.querySelectorAll()*", Mozilla Developer Network, 22 oct. 2018.
En ligne, disponible sur <https://developer.mozilla.org/en-US/docs/Web/API/Element/querySelectorAll>
[consulté le 7 fév. 2019]

[DOM2 Core], p.46

A. LE HORS, PH. LE HÉGARET, L. WOOD *et al.*, "*Document Object Model (DOM) Level 2 Core Specification*", W3C Recommendation, 13 nov. 2000.
En ligne, disponible sur <http://www.w3.org/TR/DOM-Level-2-Core/>
[consulté le 1 déc. 2015]

[DOM2 HTML], p.46

J. STENBACK, PH. LE HÉGARET, A. LE HORS (ÉD), "*Document Object Model (DOM) Level 2 HTML Specification*", W3C Recommendation, 9 jan. 2003.
En ligne, disponible sur <http://www.w3.org/TR/DOM-Level-2-HTML/>
[consulté le 1 déc. 2015]

[Selectors API], p.47

A. VAN KESTEREN, L. HUNT (ÉD), "*Selectors API Level 1*", W3C Recommendation, 21 fév. 2013.
En ligne, disponible sur <http://www.w3.org/TR/selectors-api/>
[consulté le 1 déc. 2015]

[DOM2 Events], p.52

T. PIXLEY (ÉD), "*Document Object Model (DOM) Level 2 Events Specification*", W3C Recommendation, 13 nov. 2000.
En ligne, disponible sur <http://www.w3.org/TR/DOM-Level-2-Events/>
[consulté le 1 déc. 2015]

[hsl_todo.html], p.55

D. MULLER, "*Tableaux hsl*", 19 fév. 2013, 1 p.
En ligne, disponible sur http://dmolinarius.github.io/demofiles/elc-d3/cours1/hsl_todo.html
[consulté le 7 fév. 2019]