

Version numérique

OFFERTE !

www.editions-eni.fr

JavaScript

pour l'intégrateur web

Créer des sites web
dynamiques et interactifs

Christophe AUBRY

Fichiers complémentaires
à télécharger



Handlebars
Mustache
HTML5
templates
librairies
CSS
JavaScript
Handlebars



JavaScript pour l'intégrateur web*Créer des sites web dynamiques et interactifs*

JavaScript fait partie des trois langages fondateurs du web avec l'HTML (Hypertext Markup Language) pour déterminer la structure des pages et les CSS (Cascading Style Sheets) pour concevoir la mise en forme et la mise en page. Il permet de concevoir des sites dynamiques et interactifs ; il possède l'avantage d'être reconnu nativement par tous les navigateurs web et d'être rapide à interpréter.

Ce livre est destiné aux intégrateurs web qui connaissent déjà l'HTML et les CSS et qui veulent optimiser le dynamisme et l'interactivité de leur site à l'aide du langage JavaScript.

Les premiers chapitres sont consacrés aux bases du langage JavaScript. Vous y apprendrez à insérer du JavaScript dans vos pages et vous découvrirez les règles de la **syntaxe**. Vous apprendrez à utiliser les données, les variables, les fonctions, les tableaux et aborderez la notion d'**objets JavaScript**.

Vous découvrirez ensuite le concept de DOM (Document Object Model) et vous apprendrez à l'utiliser pour accéder aux éléments constitutifs de vos pages que vous pourrez rendre interactifs grâce aux événements de souris, par exemple. Avec toutes ces notions, vous serez en mesure d'exploiter les CSS avec le JavaScript en créant de l'**interactivité** pour les visiteurs du site.

Un chapitre est consacré à des bibliothèques JavaScript qui seront une aide précieuse pour rendre dynamiques et interactives vos pages web, dans les domaines du design, de l'animation mais aussi des formulaires.

Un chapitre aborde l'apprentissage de deux moteurs de rendu JavaScript, Mustache et Handlebars, qui permettent de créer des templates JavaScript pour afficher des données formatées en JSON (JavaScript Object Notation).

Ce livre se termine par l'étude de l'**API Web Storage** qui permet de stocker des données saisies par l'utilisateur dans les navigateurs afin de les exploiter...

Christophe AUBRY

Responsable pédagogique dans un centre de formation et formateur sur les technologies Web et les Arts graphiques pendant plus de quinze ans, **Christophe AUBRY** est aujourd'hui dirigeant de la société netPlume spécialisée dans la rédaction pédagogique, la formation vidéo et la création de sites internet. Auteur de nombreux livres aux Editions ENI notamment sur Drupal, WordPress, Dreamweaver, HTML et CSS, il élargit son expertise dans l'utilisation des CMS par une veille technologique assidue.

Le JavaScript

1. Une rapide histoire du JavaScript

Ce langage est né en 1995 sous la houlette de **Brendan Eich**, qui travaillait chez **Netscape**. La légende dit que Eich créa ce langage en une dizaine de jours seulement ! Le projet initial fut baptisé **Mocha**.

Initialement, Brendan Eich développe le langage **LiveScript** qui est fait pour être utilisé côté serveur. En 1996, avec la sortie de la version 2 du navigateur de Netscape, **Netscape Navigator**, ce langage fut intégré au moteur du navigateur et nommé **JavaScript**.

Le point fort du JavaScript est qu'il n'a pas besoin d'être traduit par les navigateurs pour y être interprété et exécuté. Non, JavaScript est immédiatement disponible, comme les autres ressources de type texte et image, pour construire les pages web. JavaScript est exécuté par le navigateur.

Très rapidement, Microsoft créa un clone parfait du JavaScript qu'il nomma **JScript** et qui fut intégré à **Internet Explorer** dans sa version 3.

Ensuite, en 1997, JavaScript fut soumis à l'**ECMA** (European Computer Manufacturers Association) pour y être standardisé sous le nom d'**ECMAScript**. Actuellement, c'est toujours l'ECMA qui édicte ce standard sous le nom de **Standard ECMA-262**. La dernière version date de juin 2018 et c'est la 9e édition. C'est pour cela que vous trouverez très souvent le terme de **ES9** (**ES** pour **ECMAScript** et **9** pour **9e édition**). Cette dernière version, au moment de l'écriture de ce livre (juillet 2018), est visible à cette URL :

<https://www.ecma-international.org/publications/standards/Ecma-262.htm>, aux formats HTML et PDF.

Vous devez bien comprendre que JavaScript n'est qu'une implémentation de l'**ECMAScript 262**, côté navigateur, par la fondation Mozilla. De la même manière, le langage de script d'Adobe, l'**ActionScript**, est aussi une implémentation de l'ECMAScript 262.

2. Les moteurs JavaScript des navigateurs

Pour afficher les pages web utilisant du JavaScript, les navigateurs n'ont besoin que d'un moteur de rendu. C'est ce moteur qui va interpréter le code JavaScript et afficher les informations obtenues. Ce qu'il faut aussi bien comprendre, c'est que chaque navigateur possède son propre moteur de rendu JavaScript et que chaque navigateur est libre d'interpréter le code JavaScript comme il le veut. Bien sûr, cette vision est un peu sombre, car actuellement les navigateurs tendent, en réalité, à avoir un rendu similaire pour les standards du Web, avec les langages fondateurs que sont le HTML, les CSS et JavaScript.

Le navigateur **Google Chrome** utilise le moteur JavaScript **V8**, **Apple Safari** utilise **JavaScriptCore** (connu aussi sous les noms **SquirrelFish** et **Nitro**), **Mozilla Firefox** utilise **Rhino** et **SpiderMonkey** et **Microsoft Internet Explorer** utilise **Chakra**.

Attention, il convient de toujours avoir à l'esprit que les internautes peuvent avoir des navigateurs d'anciennes générations qui ne reconnaîtront pas forcément les toutes dernières avancées de la

dernière version du JavaScript. De plus, les internautes peuvent interdire l'exécution du JavaScript dans leur navigateur.

Les objectifs du JavaScript

1. Dynamiser les pages web

JavaScript a pour simple objectif initial de dynamiser les pages web, en y ajoutant des interactions avec les visiteurs. Les premiers développeurs qui développaient en JavaScript affichaient dans les pages web des menus déroulants, des fenêtres surgissantes (appelées maintenant des popups), voire des flocons de neige à Noël. Actuellement, ces effets semblent bien banals et un peu d'un autre âge, mais à l'époque, c'était attrayant !

De nos jours, JavaScript s'insère en tant que constructeur dynamique des pages web, à côté de la structure, définie par le **HTML**, et la mise en forme et la mise en page, définies par les **CSS**.

JavaScript va permettre d'interagir et de contrôler les actions des visiteurs. Cela va du contrôle des saisies au formulaire, aux diaporamas, jusqu'au déplacement d'objets dans une page.

2. Le JavaScript côté serveur et côté client

Depuis quelques années déjà, JavaScript s'est très fortement développé. Mais si le langage reste le même pour tous, il est utilisé par deux types de profils différents.

D'un côté, vous avez le développement d'applications côté serveur, avec des frameworks très réputés, et pas toujours faciles d'accès, comme **Ember** (<https://emberjs.com>), **React** (<https://reactjs.org>), **Angular** (<https://angular.io>) ou **Vue** (<https://vuejs.org>). Ces frameworks sont faits et sont utilisés par des développeurs purs.

De l'autre côté, vous avez le développement de sites web, de pages web qui s'affichent dans les navigateurs. C'est le côté client du développement. C'est un autre profil de développeurs qui va utiliser cette facette du JavaScript. C'est un profil intégrateur qui, avec le HTML et les CSS, va développer des sites web, des pages web dynamiques et interactives. C'est bien sûr ce type de développement que nous allons aborder dans cet ouvrage.

Les prérequis

Pour suivre au mieux le déroulement de ce livre et comprendre les exemples dispensés, vous devez être à l'aise avec les langages **HTML5** et **CSS3**. Vous le savez, HTML5 gère la structure des pages web et les CSS s'occupent de la mise en forme et de la mise en page.

Les logiciels de développement

Pour développer avec efficacité et rapidité, il vous faudra utiliser un outil de développement, un logiciel pour rédiger, pour coder vos pages web et vos scripts. Vous avez un certain nombre de solutions à votre disposition. Citons ces solutions qui sont multiplateformes :

- **Adobe Brackets** : <http://brackets.io>
- **Sublime Text** : <https://www.sublimetext.com>
- **Microsoft Visual Studio Code** : <https://code.visualstudio.com>

Ces trois outils majeurs possèdent des qualités et des points d'amélioration, ils sont extensibles avec des plugins et personnalisables selon vos habitudes de travail.

Les outils dans les navigateurs

Une fois que votre code JavaScript, HTML et CSS sera rédigé, il faudra tester vos pages web dans votre navigateur. Et il faudra certainement le déboguer pour trouver les erreurs de code. Pour ce faire, tous les navigateurs permettent d'afficher le code source généré et la structure HTML/CSS des pages créées, mais aussi d'afficher la console JavaScript.

- Avec **Apple Safari**, il faut aller dans les **Préférences**, dans la catégorie **Avancés** et cocher l'option **Afficher le menu Développement dans la barre des menus**. Vous aurez alors un menu **Développement** vous donnant accès à tous les outils voulus.
- Avec **Google Chrome**, dans le menu **Afficher**, choisissez **Options pour les développeurs**. Vous y trouverez tous les outils de développement et de débogage.
- Avec **Mozilla Firefox**, dans le menu **Outils, Développement web**, choisissez **Outils de développement**.
- Avec **Microsoft Edge**, dans le menu **Paramètres**, etc., choisissez **Outils de développement F12**.

Insérer du code JavaScript

1. Utiliser la bonne méthode

Pour ajouter de l'interaction utilisateur dans vos pages web, il faut saisir du code JavaScript, c'est une évidence. La question est donc : où insérer ce code ? En simplifiant, vous avez plusieurs solutions :

- Vous pouvez insérer tout le code JavaScript directement dans la page web, au sein d'un élément HTML `<script>`.
- Vous pouvez insérer votre code JavaScript dans un élément HTML, un bouton par exemple, qui va détecter un événement de souris l'affectant. Le code sera exécuté lors de la détection de cet événement utilisateur.

- Vous pouvez insérer tout le code JavaScript dans un fichier **.js** et vous liez ce fichier aux pages HTML voulues, aussi avec un élément HTML `<script>`.

Les deux premières méthodes sont pratiques pour les petits scripts qui ne doivent être utilisés qu'une seule fois, dans une seule page. Les modifications à apporter aux scripts pourront se faire facilement et rapidement au sein de la page HTML.

La troisième solution est bien sûr nettement plus répandue. Elle permet de centraliser les scripts dans des fichiers **.js** bien distincts. Séparer les langages est toujours préférable. De plus, ces fichiers pourront être liés à plusieurs pages HTML. Ainsi, la maintenance de ces scripts sera centralisée en un même endroit, ce qui sera plus rapide et plus efficace.

2. Insérer un script dans la page HTML

Dans ce premier exemple, nous allons ajouter un script JavaScript extrêmement simple. L'objectif est juste d'aborder la syntaxe et la structure des scripts.

Voici la structure HTML de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p>Voici un paragraphe</p>
  </body>
</html>
```

Nous souhaitons ajouter un script qui fasse appel à une simple fonction JavaScript d'alerte.

Pour ce faire, utilisons l'élément HTML `<script>`, placé juste avant la balise de fin de l'élément `</body>` :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p>Voici un paragraphe</p>
    <script>
      alert ("Bonjour tout le monde !");
    </script>
  </body>
</html>
```

Étudions la syntaxe :

- Nous avons la balise d'ouverture de l'élément `<script>`.
- C'est la fonction JavaScript `alert` que nous utilisons.
- Cette fonction possède une parenthèse ouverte et une fermée.
- Dans ces parenthèses, nous saisissons le message d'alerte à afficher, entre guillemets ".

- Enfin, la ligne de code se termine par un point-virgule ;.

Dans cet exemple, le script se déclenche tout seul, sans intervention de l'utilisateur. Le navigateur parcourt et interprète toute la page, de haut en bas. Quand le navigateur arrive dans l'élément `<script>`, il l'analyse et l'interprète aussitôt.

Voici l'affichage obtenu dans un navigateur :



L'exemple à télécharger est dans le dossier **Chapitre-02-A/exemple-01.html**.

3. Associer un script à un bouton

Nous pouvons aussi associer un script à un bouton ; c'est alors l'utilisateur qui va déclencher l'exécution de ce script par une action sur ce bouton. Dans cet exemple, l'événement détecté sera le clic de la souris sur le bouton. Donc ce n'est plus le navigateur qui va exécuter lui-même le script.

Voici cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p>Voici un bouton</p>
    <button id="monBouton" onclick="alert('Bonjour tout le monde !');">
      Cliquez-moi !
    </button>
  </body>
</html>
```

Décrivons cet exemple simple :

- Avec l'élément HTML `<button>`, nous créons un bouton.
- Ce bouton gère l'événement `onclick`, qui permet de savoir si l'utilisateur clique sur ce bouton avec sa souris.
- Si cet événement est détecté, le code JavaScript déclenche l'action, qui est ici une simple alerte.
- Remarquez que le texte de l'alerte est placé entre cote pour éviter les erreurs avec les guillemets de l'attribut `onclick`.

L'affichage obtenu est le même que précédemment.

L'exemple à télécharger est dans le dossier **Chapitre-02-A/exemple-02.html**.

4. Lier un script à une page

Cette fois, nous allons externaliser le script dans un fichier **.js** et lier ce fichier à notre page **.html** avec l'élément `<script>`.

Voici la structure HTML de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p>Voici un paragraphe</p>
    <script src="script-simple.js"></script>
  </body>
</html>
```

Nous avons une structure identique à celle du premier exemple. La différence est l'utilisation de l'élément HTML `<script>` qui est vide de tout contenu et qui possède l'attribut `src` pour indiquer la source du script, c'est-à-dire le chemin d'accès au fichier **.js**. L'attribut `src` indique que le script se trouve dans le fichier **script-simple.js**, qui est enregistré au même niveau que le fichier **.html**.

Voyons maintenant le contenu du fichier **script-simple.js** :

```
alert("Bonjour tout le monde !");
```

Le script est identique au premier exemple. La seule différence est qu'il est placé dans un fichier JavaScript défini en dehors du fichier HTML.

L'affichage obtenu est strictement identique aux exemples précédents.

L'exemple à télécharger est dans le dossier **Chapitre-02-A/exemple-03.html**.

Appliquer des règles de syntaxe

Abordons quelques règles de syntaxe pour bien écrire du JavaScript.

La première règle, qui est une obligation, est de respecter la casse. JavaScript fait la différence entre les majuscules et les minuscules. Par exemple, des fonctions nommées `maFonction`, `MaFonction` et `mafonction` ne sont pas les mêmes pour JavaScript. En effet, elles ont des casses différentes pour les lettres M et F. Vous devez donc être très vigilant sur ce point.

La deuxième règle est moins stricte. En partant du principe que nous plaçons une instruction JavaScript par ligne, chaque fin de ligne se termine par le caractère point-virgule `;`. Cette règle n'est pas absolue, mais en la respectant strictement, vous éviterez toute erreur où le point-virgule est obligatoire et vous aurez un code plus propre et plus lisible.

Une troisième règle, non obligatoire, indique qu'il vaut mieux avoir une ligne par instruction. Par exemple, nous pourrions parfaitement avoir deux fonctions `alert` dans la même ligne, si elles sont bien séparées par un point-virgule :

```
alert("Premier message");alert("Deuxième message");
```


Cette ligne sera parfaitement interprétée par les moteurs de rendu des navigateurs. Mais elle est peu lisible, et s'il y a une erreur dans cette ligne, vous ne saurez pas dans quelle fonction alert rechercher, puisque vous en avez deux !

La quatrième règle concerne les espaces. Vous pouvez insérer autant d'espaces que vous le souhaitez, ils ne seront pas interprétés. Ces lignes d'exemple sont interprétées de manière équivalente :

```
alert("Premier message");
alert ( "Premier message" ) ;
alert (      "Premier message"      )      ;
```

C'est à vous d'obtenir la syntaxe la plus claire et la plus lisible, pour vous et pour les développeurs qui pourront modifier vos pages par la suite.

Pour terminer, vous devez penser à commenter vos scripts, pour vous ou pour les autres développeurs qui vous succéderont. Les commentaires vont permettre de décrire votre code, d'expliquer ce que vous faites et ainsi le rendre plus lisible par tous. Bien sûr, ces commentaires seront ignorés par les moteurs de rendu des navigateurs. Usuellement, les développeurs insèrent les commentaires avant les instructions qu'ils décrivent.

Les commentaires sur une seule ligne s'insèrent en commençant par deux caractères barre oblique : //. Les commentaires sur plusieurs lignes s'insèrent avec les caractères /* au début et */ à la fin.

Exemples :

```
// Création de la fonction
function monAlerte(){
    alert("Bonjour tout le monde !");
};

/* Appel de la fonction
créée précédemment */
monAlerte();
```

Placer le code JavaScript dans la page

Abordons maintenant un point très important. Dans le cas de figure où vous décidez d'insérer votre code JavaScript au sein même de la page HTML, il faut déterminer avec soin où le placer. Cela va avoir des implications importantes.

Premier point : en simplifiant, nous pouvons dire que dans un navigateur, le code, que ce soit du HTML, des CSS ou du JavaScript, est lu et interprété de haut en bas. Donc si nous voulons agir sur un élément de la page par un script, il faut d'abord que le navigateur connaisse la structure HTML de la page, avant qu'il interprète le script qui va cibler cet élément de la page. Donc si le script est chargé avant la structure HTML, le script ne pourra tout simplement pas fonctionner, car il ne connaîtra pas l'élément HTML avec lequel il doit interagir.

Deuxième point : nous pouvons parfaitement placer des scripts dans l'élément <head> des pages HTML. Mais si nous plaçons trop de scripts dans l'élément <head>, le temps de chargement de la page va s'en trouver pénalisé.

C'est pour ces deux raisons que la méthode la plus couramment utilisée consiste à placer l'élément `<script>` juste avant la balise de fermeture `</body>` de l'élément HTML `<body>`. Cet élément `<script>` peut soit contenir le code, soit indiquer un lien vers un fichier JavaScript.

Voici la syntaxe que nous avons utilisée précédemment :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p>Voici un paragraphe</p>
    <script src="script-simple.js"></script>
  </body>
</html>
```

Dans les exemples précédents, nous avons utilisé la fonction `alert` du JavaScript. Cette fonction peut être utilisée pour constater si nos codes fonctionnent bien en plaçant des points de contrôle affichant, par exemple, la valeur d'une donnée, ou en affichant une alerte pour noter si une fonction fonctionne correctement. Cette fonction `alert`, si pratique soit-elle, n'est pas la solution idéale. En effet, dans la fenêtre d'alerte qui s'affiche, cette fonction oblige l'utilisateur à cliquer sur le bouton de confirmation (**OK** le plus souvent) et bloque l'exécution du code tant que l'utilisateur n'a pas cliqué sur ce bouton.

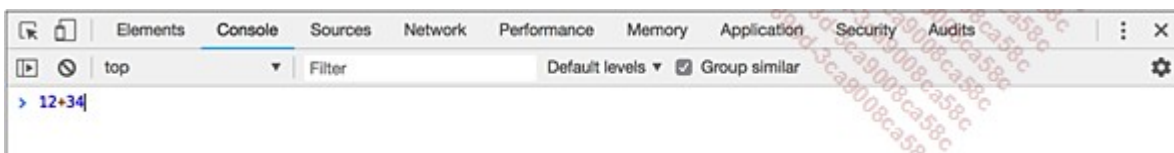
Pour générer des messages dans l'exécution du code JavaScript, il est recommandé d'utiliser la console incluse dans tous les navigateurs modernes. De plus, cette console va nous permettre de tester l'exécution du code JavaScript et elle possède les outils nécessaires pour localiser les erreurs et nous aider à les résoudre.

2. Calculer dans la console

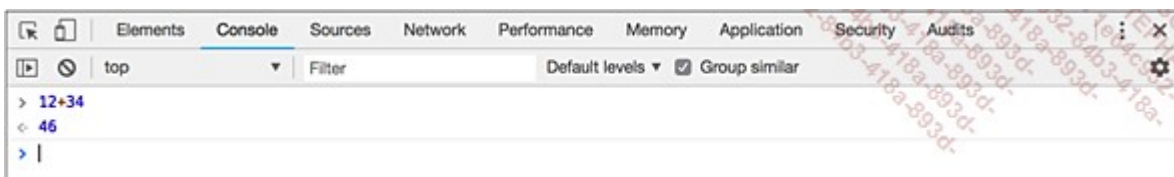
Pour appréhender la console JavaScript de votre navigateur, nous allons faire un simple calcul mathématique.

Dans votre navigateur, affichez les outils de développement et cliquez sur l'onglet **Console**.

Dans la zone de saisie, saisissez 12+34, par exemple.



Validez avec la touche [Entrée], le calcul se fait.



3. Afficher une alerte dans la console

Cette fois, nous allons afficher le message d'une alerte dans la console et non plus dans une fenêtre modale.

Créez un nouveau fichier JavaScript et enregistrez-le sous le nom de **test-console.js**, par exemple.

Dans celui-ci, saisissez ce simple code :

```
// Envoyer un message dans la console
console.log("Bonjour tout le monde !");
```

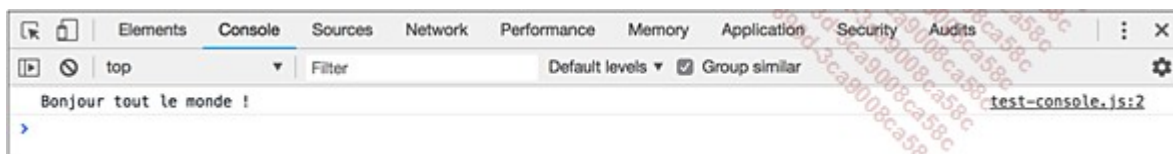
Nous utilisons la fonction JavaScript `console.log()`, qui permet d'afficher un message dans la console.

Créez un nouveau fichier HTML et saisissez-y ce code :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p>Voici un paragraphe</p>
    <script src="test-console.js"></script>
  </body>
</html>
```

Ouvrez ce fichier HTML dans votre navigateur.

Affichez la console JavaScript et vous voyez le message apparaître :



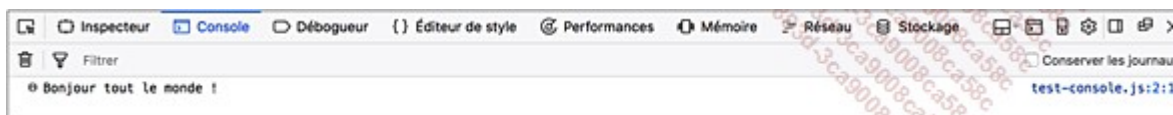
L'exemple à télécharger est dans le dossier **Chapitre-02-D/exemple-01.html**.

4. Les autres messages pour la console

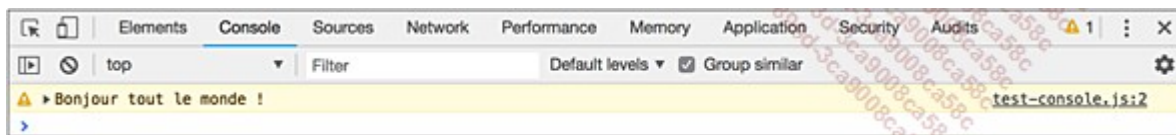
La fonction console peut s'utiliser avec d'autres suffixes qui permettent de personnaliser l'affichage du message.

- `console.debug()` permet d'afficher un message de débogage, avec une autre couleur.
- `console.info()` permet d'afficher le message sous la forme d'une information, avec une petite icône **i** sur certains navigateurs.

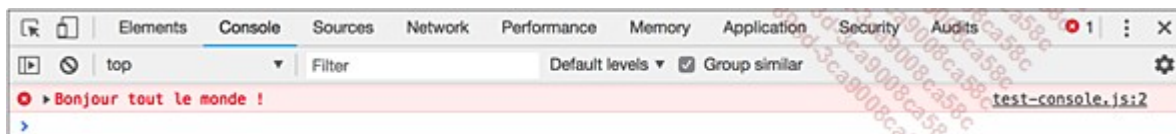
Voici l'affichage obtenu avec **Mozilla Firefox** :



- `console.warn()` permet d'afficher le message avec une autre icône et un fond coloré jaune. À nouveau, l'affichage peut varier selon le navigateur.



- `console.error()` permet d'afficher le message avec une autre icône et un fond coloré rouge.



Le message affiché dans la console est toujours le même, sans différenciations techniques. C'est juste son affichage qui peut être personnalisé avec ces suffixes différents, selon le navigateur que vous utilisez.

Les données dans le code JavaScript

1. Définir les types de données

Lorsque vous allez coder en JavaScript, vous allez utiliser des données. Ces données sont des informations de différents types. Vous avez des données numériques, comme des nombres : **8**, **14**, **123**... Vous pourrez aussi utiliser du texte, des caractères, comme : **Février**, **Nantes**, **Durand**... D'autres types de données sont peut-être moins parlants de prime abord, comme le type booléen qui détermine si une donnée est vraie (true) ou fausse (false).

Donc, en codant avec JavaScript, vous serez amené à utiliser ces types de données. Notez bien que JavaScript est un langage de programmation qui est "faiblement typé". Cela veut dire qu'il ne sera pas strictement nécessaire de définir le type de données avant d'utiliser celles-ci. Par exemple, si vous utilisez la donnée **8**, JavaScript détermine qu'il s'agit d'un chiffre et non pas du caractère 8. De même, si vous utilisez la donnée **Durand**, JavaScript l'utilise en tant que chaîne de caractères. Cela facilite l'utilisation de ces données, même si les développeurs les plus "puristes" peuvent parfaitement dire que JavaScript manque de rigueur, de précision et de bases solides pour parfaire le code.

JavaScript n'utilise que cinq types de données, nommés primitifs, ce qui simplifie la création des scripts.

2. Le type Number

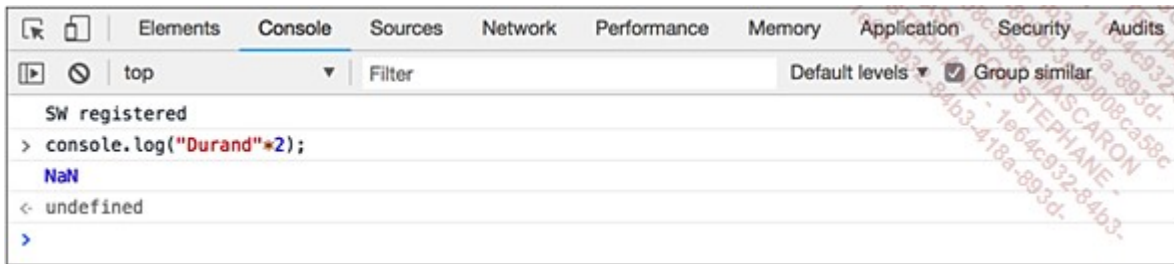
Le type de données **Number** définit les données de type numérique, quelle que soit la valeur numérique : -12 345, 123, 8.123. Attention, dans les langages de programmation, le séparateur décimal est un point et non une virgule, comme vous pouvez le voir dans le dernier exemple précédent. Toutes ces valeurs sont des données de type numérique. Notez bien aussi que le type **Number** ne fait pas la différence entre les nombres entiers et les nombres décimaux.

Un point important à noter est l'utilisation du type de données particulier qu'est **NaN**. **NaN** veut dire **Not a Number**, "pas un nombre". Cette indication peut être affichée lorsque vous voulez effectuer des calculs avec du texte, par exemple.

Ainsi, si dans la console vous saisissez :

```
console.log("Durand"*2);
```

Vous obtiendrez une erreur avec l’affichage de **NaN**.



Il n’est pas possible de multiplier par 2 du texte. JavaScript sait que **Durand** n’est pas un nombre mais une chaîne de caractères, donc il ne peut faire le calcul.

Avec les données de type **Number**, vous allez bien sûr pouvoir faire des calculs avec les opérateurs mathématiques classiques :

- Addition avec l’opérateur `+`. Exemple : `1+2` donne le résultat 3.
- Soustraction avec l’opérateur `-`. Exemple : `3-2` donne le résultat 1.
- Multiplication avec l’opérateur `*`. Exemple : `3*2` donne le résultat 6.
- Division avec l’opérateur `/`. Exemple : `8/2` donne le résultat 4.
- Avec l’opérateur `++`, vous ajoutez 1 à un nombre. C’est une incrémentation. Exemple : `4++` donne le résultat 5.
- Avec l’opérateur `--`, vous soustrayez 1 à un nombre. C’est une décrémentation. Exemple : `4--` donne le résultat 3.
- Avec l’opérateur `%`, vous obtenez le reste d’une division, vous calculez le modulo. Exemple : `12%5` donne le résultat 2.

3. Le type String

Le type de données **String** définit les données de type chaîne de caractères, c’est-à-dire une succession de lettres, de chiffres et de symboles. Les chaînes de caractères sont toujours indiquées entre guillemets, "Durand", ou entre simples quotes, 'Dupond 44'.

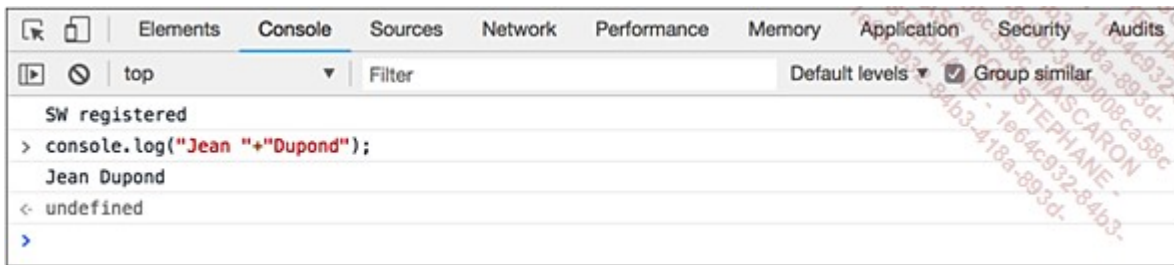
Mettre "bout à bout" deux chaînes de caractères s’appelle une **concaténation**. Pour ce faire, il faut utiliser l’opérateur `+` qui, dans ce cas, ne fait bien sûr pas de calcul, mais concatène les deux chaînes de caractères.

Dans la console, saisissez cette simple ligne de code :

```
console.log("Jean "+"Dupond");
```

Validez avec la touche [Entrée].

Voici le résultat obtenu : **Jean Dupond**.



Attention, certains caractères ont une signification particulière en JavaScript, comme les guillemets ou les apostrophes. Donc si vous devez les utiliser dans une chaîne de caractères, vous devez saisir un caractère spécial avant, pour indiquer au script leur utilisation spécifique. C'est ce qui s'appelle un caractère d'échappement. En JavaScript, utilisez la barre oblique inversée, couramment appelée antislash.

Si vous indiquez cette commande dans la console :

```
console.log('Jean d'Aubry');
```

Vous aurez une erreur, car pour JavaScript la chaîne de caractères s'arrête juste après la lettre d, avec la deuxième simple quote.

Vous pouvez alors indiquer :

```
console.log('Jean d\'Aubry');
```

Avec cette syntaxe, vous n'aurez pas d'erreur.

Vous pouvez aussi utiliser le caractère d'échappement pour insérer des caractères spéciaux :

- `\n` insère une nouvelle ligne.
- `\r` insère un retour chariot.
- `\t` insère une tabulation.
- `\b` insère un retour arrière.
- `\f` insère un saut de page.

4. Le type Undefined

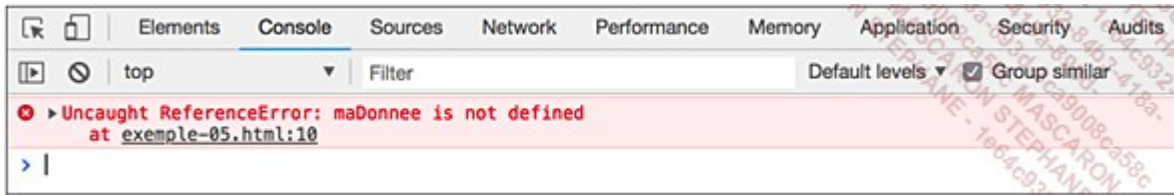
Le type de données **Undefined** désigne les types de données qui ne sont pas définis dans un script. Par exemple, vous souhaitez afficher le résultat d'un calcul dans un élément HTML. Si ce dernier n'est pas défini dans le script, la console nous renvoie alors ce type d'erreur.

Voyons un autre exemple de ce type d'erreur. Voici le code de cet exemple simple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p id="affiche"></p>
    <script>
      document.getElementById("affiche").innerHTML=maDonnee;
    </script>
```

```
</body>
</html>
```

Voici l'affichage obtenu :



Expliquons l'origine de cette erreur :

- Le script cherche dans la page l'élément dont l'identifiant unique est affiché : `document.getElementById("affiche")`.
- Puis, le script cherche à y afficher une donnée : `innerHTML`.
- Et cette donnée est `maDonnee`.

Mais le problème est que cette donnée, nommée `maDonnee`, n'existe pas, elle n'a jamais été définie dans le script. Donc le script ne la connaît pas. D'où l'affichage de cette erreur dans la console.

5. Le type Null

Le type **Null** indique qu'une donnée ne possède pas de valeur. Attention, cela n'indique pas la valeur **0**. Non, **Null** est une "non-valeur". C'est intéressant pour définir une donnée dont nous ne connaissons pas encore la valeur, ou pour ne pas attribuer de valeur à une donnée à un moment précis du script.

6. Le type Boolean

Les données booléennes ne peuvent avoir que deux valeurs possibles : `true`, pour vraie, ou `false`, pour fausse.

Le terme "booléen" trouve son origine dans le nom d'un mathématicien anglais nommé George Boole, qui développa l'algèbre logique. C'est pour cela que le terme de "logique booléenne" est très souvent utilisé.

En JavaScript, nous pouvons faire des tests qui renverront soit une réponse vraie, `true`, soit une réponse fausse, `false`.

Les variables

1. L'objet variable

Comme leur nom l'indique, les variables permettent de stocker des données qui peuvent varier. Au cours de l'exécution du script, la donnée, la valeur d'une variable, peut varier. Nous pouvons par exemple définir une variable qui va stocker le résultat d'un calcul. Initialement, la valeur peut être **Null**, car il n'y a pas de valeur. Ensuite, la variable peut prendre une première valeur, puis, quelques

lignes plus loin, le script peut additionner une autre valeur à la valeur stockée dans la variable. Nous avons alors une nouvelle valeur stockée dans la variable.

2. Déclarer une variable

Pour utiliser une variable, il faut obligatoirement la déclarer. Pour ce faire, nous avons deux solutions à notre disposition.

La première méthode permet de créer rapidement une variable, avec cette simple syntaxe :

```
monPrenom = "Christophe";
```

Décrivons cette ligne :

- Le nom de la variable est monPrenom.
- Le signe utilisé pour stocker une valeur dans la variable est =.

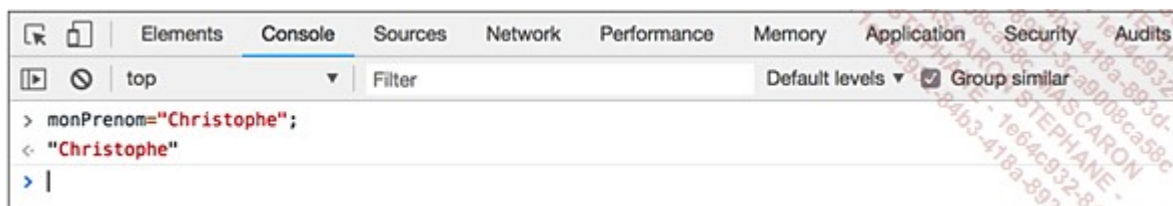


Attention, ici le signe = est le symbole de l'affectation, ou assignation, et non pas le symbole de la comparaison.

- La valeur initiale de cette variable est indiquée, dans cet exemple, entre guillemets et est Christophe.

Dans cet exemple, la variable n'est pas typée mais, du fait des guillemets, JavaScript sait qu'il s'agit d'une variable de type **String**, qui contient une chaîne de caractères. Autre remarque, cette variable possède immédiatement une valeur, qui est Christophe.

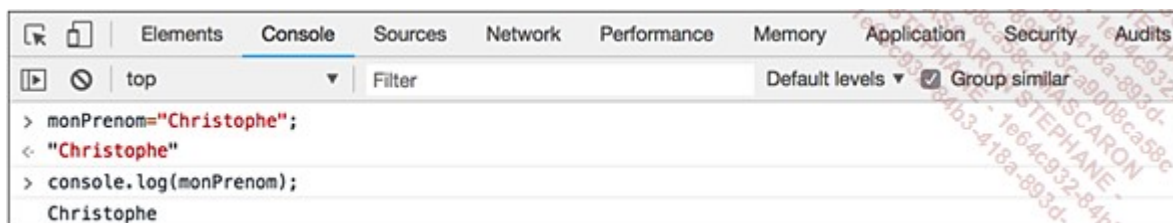
Voici la saisie de cette variable dans la console et la validation qui suit avec la touche [Entrée] :



Dans la console, nous pouvons visualiser que la variable est bien affectée d'une valeur :

```
console.log(monPrenom);
```

Voici l'affichage obtenu :



La deuxième méthode utilise le mot réservé var pour déclarer la variable, avec cette syntaxe :

```
var monPrenom;
```


Dans cet exemple, la variable n'a pas de valeur initiale. Mais nous pouvons y stocker une valeur, dès la création de la variable, avec cette syntaxe :

```
var monPremon = "Christophe";
```

Les deux syntaxes sont parfaitement valides. Mais d'une manière générale, privilégiez toujours l'utilisation du mot réservé `var`. C'est une très bonne habitude, cela permettra de supprimer des ambiguïtés et donnera un code plus facile à lire.

L'autre possibilité consiste à créer plusieurs variables dans la même ligne, avec un seul `var`, chaque variable étant séparée de la suivante par une virgule `,`.

Voici une syntaxe parfaitement valide :

```
var mon_prenom="Christophe", mon_nom="Aubry",  
mon_email="christophe@aubry.org";
```

Jusqu'à présent, la variable était implicitement de type **String**, du fait même de la présence des guillemets. Si une variable est de type **Number**, si elle doit donc stocker une valeur numérique, il suffit d'indiquer cette valeur, juste après le signe `=`.

Voici un exemple :

```
var nombre_connexion = 123;
```

3. Le nom des variables

Le nom des variables n'est pas libre, vous devez respecter des règles de nommage :

- Le nom peut commencer par une lettre en majuscule ou en minuscule.
- Le nom peut commencer par le caractère `_`.
- Le nom peut commencer par le caractère `$`.
- Le nom ne doit pas contenir d'espaces.
- Le nom ne doit pas contenir des caractères spéciaux, ni des caractères accentués ou des opérateurs mathématiques.
- Le nom ne doit pas être celui d'un mot réservé par JavaScript, comme `case`, `var`, `double`, `continue`...

Si une variable doit être composée de plusieurs mots, vous pouvez utiliser la syntaxe **Camel Case**. Le premier mot commence par une minuscule et chaque autre mot commence par une majuscule. Exemples : `monPrenom`, `dateDeNaissance`. Vous pouvez aussi séparer les mots par le caractère `_`. Exemples : `mon_prenom`, `date_de_naissance`.

4. Les opérations de calcul avancées

Nous avons vu dans une section précédente, les opérateurs mathématiques classiques que sont l'addition, la soustraction, la multiplication, la division et le modulo. Mais JavaScript nous propose d'autres opérateurs de calcul qui sont très couramment utilisés.

Ce premier exemple permet de faire une addition et de l'affecter à une même variable.

Voici la syntaxe :

```
var a = 3 ;  
a = a + 10 ;
```

Décrivons ces deux lignes :

- Dans la première ligne, nous déclarons une variable numérique, nommée a et affectée de la valeur 3.
- Dans la deuxième ligne, nous affectons à la variable a sa propre valeur initiale, 3, et nous lui ajoutons la valeur de 10.
- Nous avons donc comme résultat 13.

Ce type de calcul est extrêmement courant. Donc, pour aller plus vite, nous pouvons utiliser cette syntaxe :

```
var a = 3 ;  
a += 10 ;
```

Nous obtenons strictement le même résultat.

Nous pouvons utiliser de la même manière les autres opérateurs, -, * et /.

Autre opération très classique, c'est l'incrémentation. Elle s'utilise avec cette syntaxe :

```
var a = 0 ;  
a++ ;
```

Cela permet d'augmenter la valeur de la variable a de 1.

Vous pouvez aussi décrémenter la variable avec cette syntaxe :

```
var a = 10 ;  
a-- ;
```

5. Les opérations de comparaison

Nous venons de voir que le signe = est le symbole de l'affectation, de l'assignation d'une valeur à une variable. Si vous devez comparer deux variables, il faut utiliser deux fois le signe = pour avoir le symbole d'égalité.

Voici un exemple :

```
var a = 2, b = 3 ;  
console.log( a==b ) ;
```

La console va renvoyer false, car la valeur de la variable a n'est pas égale à celle de b.

Si vous saisissez ce code :

```
var a = 2, b = 3 ;  
console.log( a=b ) ;
```

Et que vous validez avec la touche [Entrée], la console va renvoyer 3. En effet, vous affectez la valeur de la variable b à la variable a.

Voici un autre exemple :

```
var a = 2, b = 2 ;  
console.log( a==b ) ;
```

La console va renvoyer true, car la valeur de la variable a est bien égale à celle de b, c'est-à-dire 2.

Voyons maintenant un autre exemple de comparaison que vous risquez de trouver, c'est la comparaison stricte. Cette comparaison stricte utilise le triple signe égal : ===.

Voici un premier exemple :

```
var a = 2, b = "2" ;  
console.log( a==b ) ;
```

La console va renvoyer true. En effet, la variable a contient la valeur numérique 2 et la variable b contient la chaîne de caractères 2. Nous avons donc bien une égalité, même si les variables sont de types différents, **Number** pour a et **String** pour b.

Voici un deuxième exemple :

```
var a = 2, b = "2" ;  
console.log( a===b ) ;
```

La console va renvoyer false. En effet, les deux variables contiennent la même valeur 2, mais elles sont de types différents. Donc nous n'avons pas une égalité stricte.

Vous pouvez aussi utiliser la négation de l'égalité avec l'opérateur !=, qui se lit "n'est pas égal à". Le symbole ! implique la négation de l'opérateur.

Voici un exemple :

```
var a = 2, b = 3 ;  
console.log( a!=b ) ;
```

La console renvoie true, car effectivement la valeur de la variable a n'est pas égale à la valeur de la variable b.

Pour terminer ces opérateurs de comparaison, vous avez bien sûr à votre disposition les classiques opérateurs >, >=, < et <=.

Les tests de condition

1. Créer des tests

Lorsque vous avez des variables, dont par essence même la valeur varie, il est parfois primordial de connaître la valeur d'une variable par des tests. Et en fonction de ces tests, vous devrez exécuter une action ou une autre, si la réponse à ces tests renvoie vrai (true) ou faux (false). C'est l'objectif de ces tests de condition.

Ces tests conditionnels s'utilisent avec la fonction JavaScript if(). Voici la syntaxe de base du test de condition :

```
if(test) {  
    réponse si vrai;  
}
```

Entre les parenthèses de la fonction if, nous avons le test qui est exécuté par le script. Ensuite, entre les accolades, nous indiquons ce qui doit être fait, si la réponse du test est vraie (true).

2. Les tests simples avec if

Dans ce premier exemple, nous allons réaliser un test simple sur une variable. Nous allons tester si une variable est supérieure à une valeur spécifiée.

Voici le code utilisé :

```
var a=100 ;
if(a>20){
    console.log("a est plus grand que 20");
}
```

La réponse au test est vraie, le test if renvoie true. La valeur de la variable a, qui est égale à 100, est effectivement bien supérieure à 20. Puisque c'est vrai, nous affichons dans la console un message.

Voici l'affichage obtenu :



Si nous changeons la valeur de la variable a à 10 par exemple, la console n'affichera rien, puisqu'il n'y a aucune instruction à exécuter si le test renvoie false.

```
var a=10 ;
if(a>20){
    console.log("a est plus grand que 20");
}
```

Voici l'affichage obtenu :



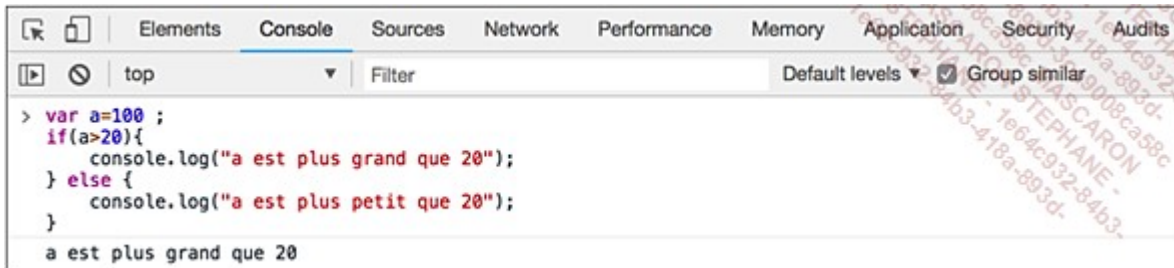
Il est donc plus judicieux d'envisager dans notre test les deux réponses possibles : soit le test renvoie vrai, soit il renvoie faux. Nous allons donc ajouter cette deuxième possibilité en insérant l'instruction else, qui se lit "sinon".

Voici le code modifié :

```
var a=100 ;
if(a>20){
    console.log("a est plus grand que 20");
} else {
    console.log("a est plus petit que 20");
}
```

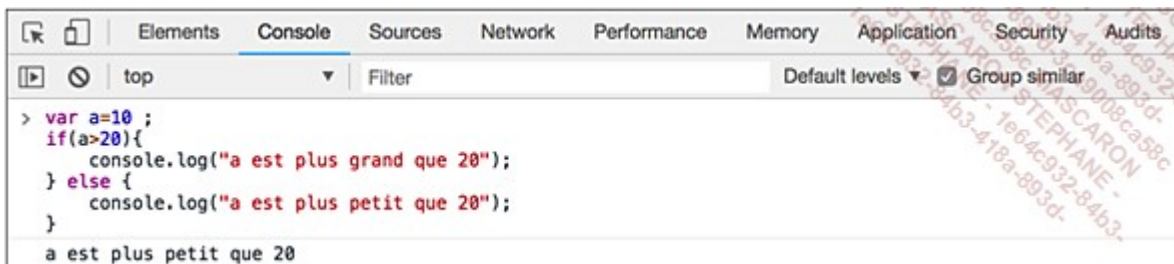
L'insertion de l'instruction else implique l'utilisation d'un deuxième couple d'accolades et ce couple d'accolades indique l'instruction qui doit être exécutée lorsque le test est faux.

Voici l'affichage obtenu avec a=100 :



```
> var a=100 ;  
if(a>20){  
  console.log("a est plus grand que 20");  
} else {  
  console.log("a est plus petit que 20");  
}  
  
a est plus grand que 20
```

Voici l'affichage obtenu avec a=10 :



```
> var a=10 ;  
if(a>20){  
  console.log("a est plus grand que 20");  
} else {  
  console.log("a est plus petit que 20");  
}  
  
a est plus petit que 20
```

Pour des tests simples, vous pourrez rencontrer cette syntaxe courte :

```
var a=100 ; a>20? console.log("a est plus grand que 20"): console.log("a est plus petit que 20") ;
```

Étudions cette syntaxe courte :

- Nous indiquons le test a>20.
- Le test est immédiatement suivi par le caractère ?.
- Nous indiquons ensuite ce qui doit être fait si le test renvoie true: console.log("a est plus grand que 20").
- L'instruction du vrai est immédiatement suivie par le caractère :.
- Nous indiquons ensuite ce qui doit être fait si le test renvoie false:console.log("a est plus petit que 20").
- La fin de la ligne se termine par un classique point-virgule.

3. Les tests logiques avec ET et OU

Dans l'exemple précédent, nous n'avons réalisé qu'un seul test. Mais il se peut que vous ayez besoin de tester plusieurs variables. On peut parfaitement imaginer de tester si un utilisateur habite dans le département du 44 et s'il est âgé de plus de 40 ans, par exemple.

Avec l'utilisation des tests multiples, vous devez répondre à la question suivante : les tests doivent-ils être vrais tous les deux, pour obtenir true, ou bien un seul test vrai suffit-il pour obtenir true ? Dans le premier cas, vous devrez utiliser une logique **ET**, dans le deuxième cas, vous devrez utiliser la logique **OU**.

Prenons un exemple : définissons trois variables, avec trois valeurs :

```
var a=12, b=36, c=22 ;
```

L'objectif est de tester si ces variables sont supérieures à 20. Nous voyons qu'une seule variable est inférieure à 20 et que les deux autres sont supérieures à 20.

Avec la logique **ET**, il faut que toutes les variables soient supérieures à 20 pour obtenir true. Si une seule variable, ou plus, n'est pas supérieure à 20, le test renvoie false.

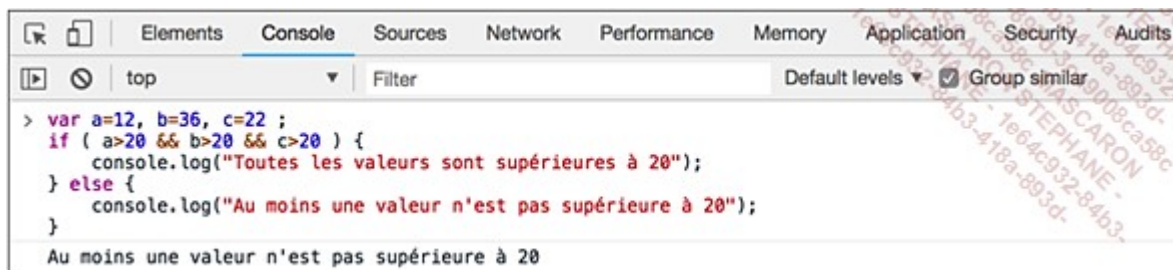
Avec la logique **OU**, il faut qu'au moins une variable soit supérieure à 20 pour obtenir true. Si toutes les variables ne sont pas supérieures à 20, le test renvoie false.

La logique **ET** s'écrit avec la double esperluette && et la logique **OU** s'écrit avec la double barre verticale (pipe en anglais) ||.

Saisissons le code de cet exemple avec la logique **ET** :

```
var a=12, b=36, c=22 ;
if ( a>20 && b>20 && c>20 ) {
    console.log("Toutes les valeurs sont supérieures à 20");
} else {
    console.log("Au moins une valeur n'est pas supérieure à 20");
}
```

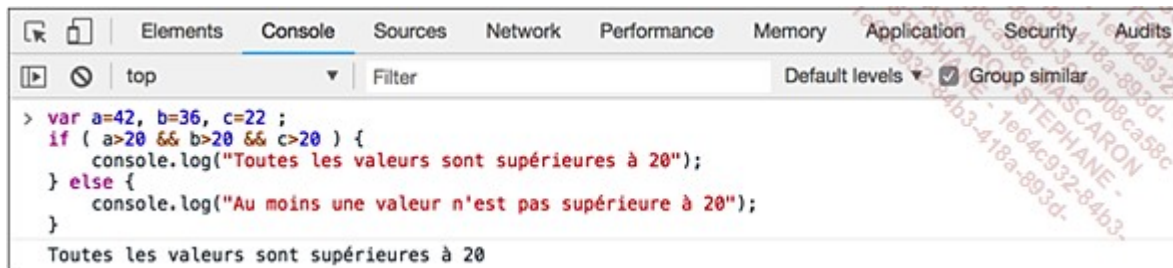
Voici l'affichage obtenu :



Voici un deuxième exemple avec la logique **ET** :

```
var a=42, b=36, c=22 ;
if ( a>20 && b>20 && c>20 ) {
    console.log("Toutes les valeurs sont supérieures à 20");
} else {
    console.log("Au moins une valeur n'est pas supérieure à 20");
}
```

Voici l'affichage obtenu :

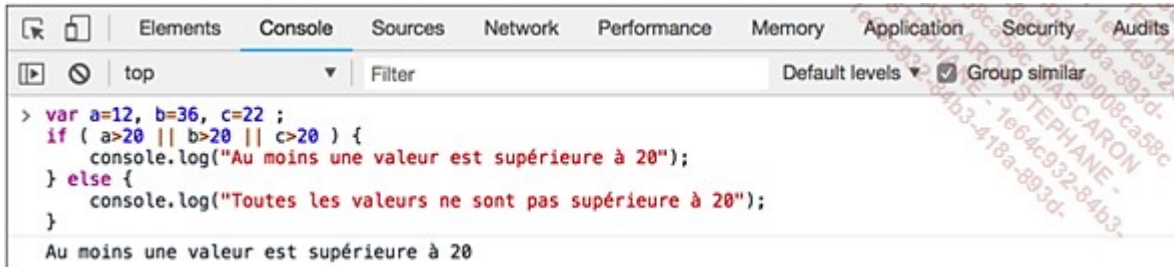


Saisissons le code cet exemple avec la logique **OU** :

```
var a=12, b=36, c=22 ;
if ( a>20 || b>20 || c>20 ) {
    console.log("Au moins une valeur est supérieure à 20");
} else {
    console.log("Toutes les valeurs ne sont pas supérieures à 20");
}
```

```
}
```

Voici l'affichage obtenu :

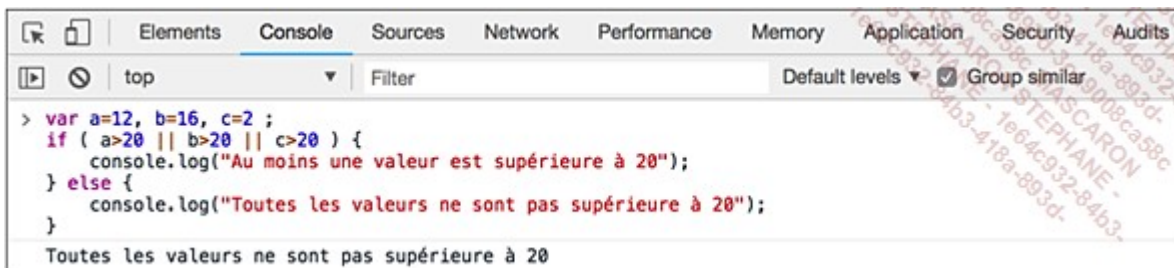


```
> var a=12, b=36, c=22 ;  
  if ( a>20 || b>20 || c>20 ) {  
    console.log("Au moins une valeur est supérieure à 20");  
  } else {  
    console.log("Toutes les valeurs ne sont pas supérieure à 20");  
  }  
  Au moins une valeur est supérieure à 20
```

Voici un deuxième exemple avec la logique **OU** :

```
var a=12, b=16, c=2 ;  
if ( a>20 || b>20 || c>20 ) {  
    console.log("Au moins une valeur est supérieure à 20");  
} else {  
    console.log("Toutes les valeurs ne sont pas supérieures à 20");  
}
```

Voici l'affichage obtenu :



```
> var a=12, b=16, c=2 ;  
  if ( a>20 || b>20 || c>20 ) {  
    console.log("Au moins une valeur est supérieure à 20");  
  } else {  
    console.log("Toutes les valeurs ne sont pas supérieure à 20");  
  }  
  Toutes les valeurs ne sont pas supérieure à 20
```

4. Les tests imbriqués avec else if

Il se peut que vous ayez besoin d'effectuer plusieurs tests successifs. Dans ce cas, vous allez utiliser des instructions `if()` imbriquées et des instructions `else`, avec cette syntaxe :

```
if (test1) {  
    vrai1 ;  
} else if (test2) {  
    vrai2 ;  
} else {  
    faux;  
}
```

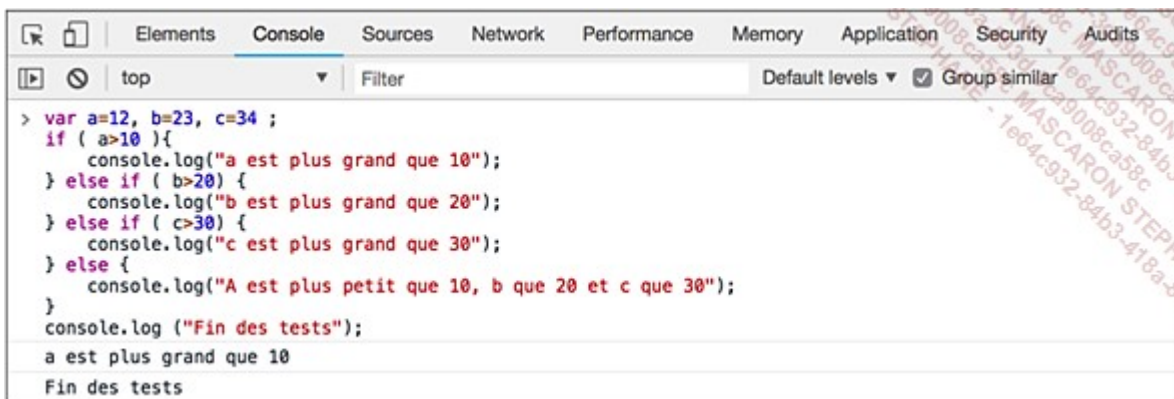
Voici un exemple :

```
var a=12, b=23, c=34 ;  
if ( a>10 ){  
    console.log("a est plus grand que 10");  
} else if ( b>20) {  
    console.log("b est plus grand que 20");  
} else if ( c>30) {  
    console.log("c est plus grand que 30");  
} else {  
    console.log("A est plus petit que 10, b que 20 et c que 30");  
}  
console.log ("Fin des tests");
```

Analysons le code :

- Nous créons trois variables numériques, affectées de trois valeurs.
- Nous faisons un premier test, if(), qui détermine si a est plus grand que 10.
- Si c'est vrai, nous affichons un message dans la console, puis nous sortons du test if() initial et nous passons à la suite du code.
- Si c'est faux, nous passons au deuxième test, else if.
- Le deuxième if() teste si b est plus grand que 20.
- Si c'est vrai, nous affichons un message dans la console, puis nous sortons du deuxième test if() et nous passons à la suite du code.
- Si c'est faux, nous passons au troisième test, else if.
- Le troisième if() teste si c est plus grand que 30.
- Si c'est vrai, nous affichons un message dans la console, puis nous sortons du troisième test if() et nous passons à la suite du code.
- Si c'est faux, nous affichons un quatrième et dernier message dans la console et nous passons à la suite du code.

Voici l'affichage obtenu avec ce premier exemple :

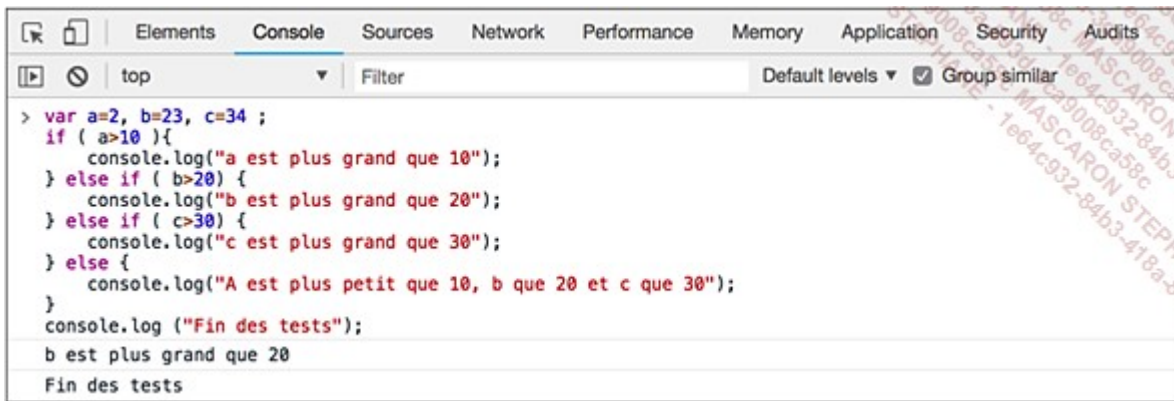


```
> var a=12, b=23, c=34 ;  
  if ( a>10 ){  
    console.log("a est plus grand que 10");  
  } else if ( b>20 ) {  
    console.log("b est plus grand que 20");  
  } else if ( c>30 ) {  
    console.log("c est plus grand que 30");  
  } else {  
    console.log("A est plus petit que 10, b que 20 et c que 30");  
  }  
  console.log ("Fin des tests");  
a est plus grand que 10  
Fin des tests
```

Voici le deuxième exemple, en modifiant la valeur de la variable a :

```
var a=2, b=23, c=34 ;  
if ( a>10 ){  
  console.log("a est plus grand que 10");  
} else if ( b>20 ) {  
  console.log("b est plus grand que 20");  
} else if ( c>30 ) {  
  console.log("c est plus grand que 30");  
} else {  
  console.log("A est plus petit que 10, b que 20 et c que 30");  
}  
console.log ("Fin des tests");
```

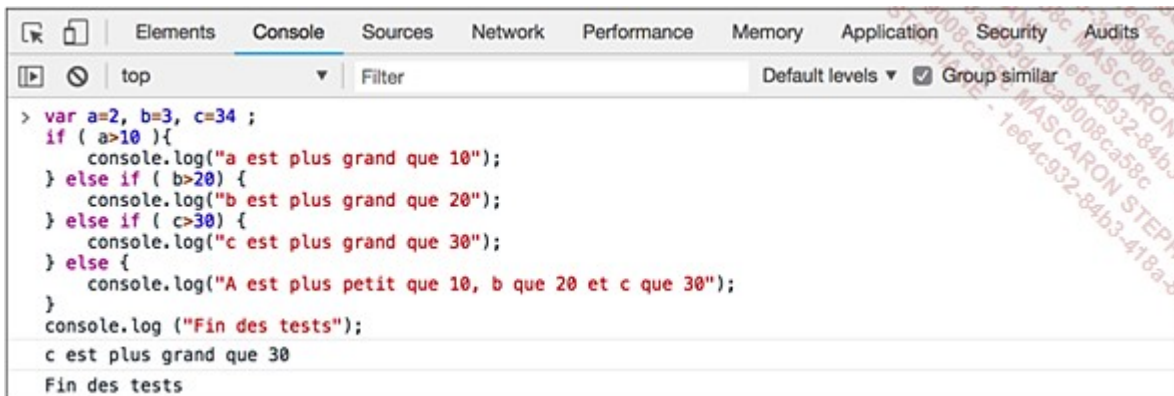
Voici l'affichage obtenu avec ce deuxième exemple :



Voici le troisième exemple, en modifiant la valeur de la variable b :

```
var a=2, b=3, c=34 ;  
if ( a>10 ){  
  console.log("a est plus grand que 10");  
} else if ( b>20) {  
  console.log("b est plus grand que 20");  
} else if ( c>30) {  
  console.log("c est plus grand que 30");  
} else {  
  console.log("A est plus petit que 10, b que 20 et c que 30");  
}  
console.log ("Fin des tests");
```

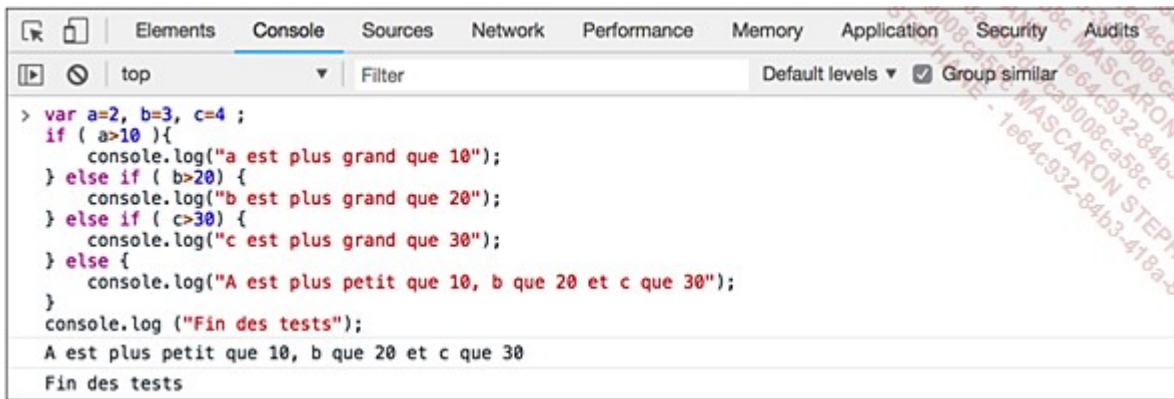
Voici l'affichage obtenu avec ce troisième exemple :



Voici le quatrième exemple, en modifiant la valeur de la variable c :

```
var a=2, b=3, c=4 ;  
if ( a>10 ){  
  console.log("a est plus grand que 10");  
} else if ( b>20) {  
  console.log("b est plus grand que 20");  
} else if ( c>30) {  
  console.log("c est plus grand que 30");  
} else {  
  console.log("A est plus petit que 10, b que 20 et c que 30");  
}  
console.log ("Fin des tests");
```

Voici l'affichage obtenu avec ce quatrième exemple :

A screenshot of a web browser's developer console. The 'Console' tab is active, showing a series of log messages. The code being executed is a series of if-else statements checking the values of variables a, b, and c. The output shows that a is not greater than 10, b is not greater than 20, and c is not greater than 30, so the final log message is 'A est plus petit que 10, b que 20 et c que 30'. The console also shows 'Fin des tests' at the end.

```
> var a=2, b=3, c=4 ;
if ( a>10 ){
  console.log("a est plus grand que 10");
} else if ( b>20 ) {
  console.log("b est plus grand que 20");
} else if ( c>30 ) {
  console.log("c est plus grand que 30");
} else {
  console.log("A est plus petit que 10, b que 20 et c que 30");
}
console.log ("Fin des tests");
A est plus petit que 10, b que 20 et c que 30
Fin des tests
```

5. Les tests multiples avec switch

Dans le cas où vous avez beaucoup de tests à effectuer, l'imbrication des tests successifs peut être très rapidement ingérable. Il faut alors changer de stratégie et utiliser l'instruction `switch()`. Cette instruction utilise aussi des tests conditionnels, mais de manière beaucoup plus souple et facile à utiliser.



Attention, notez bien que l'instruction `switch()` ne fonctionne qu'avec des conditions d'égalité, mais pas avec d'autres opérateurs comme inférieur, supérieur...

Dans cet exemple, nous allons tester les valeurs d'une même variable de type texte (String).

Voici le code utilisé :

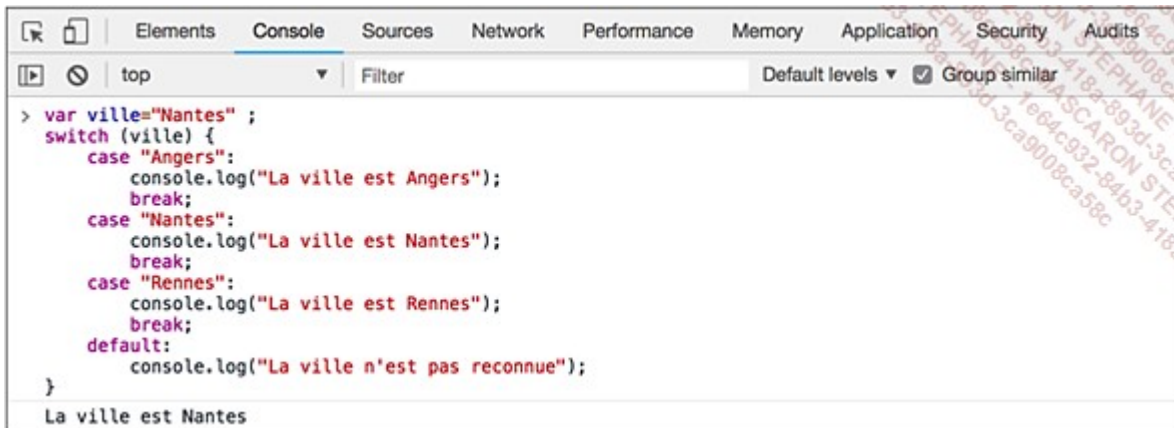
```
var ville="Nantes" ;
switch (ville) {
  case "Angers":
    console.log("La ville est Angers");
    break;
  case "Nantes":
    console.log("La ville est Nantes");
    break;
  case "Rennes":
    console.log("La ville est Rennes");
    break;
  default:
    console.log("La ville n'est pas reconnue");
}
```

Analysons ce code :

- Nous définissons une variable nommée `ville`, ayant la valeur `Nantes`.
- Nous utilisons l'instruction `switch`, avec entre parenthèses la variable `ville` qui est à tester.
- Le mot-clé `case` indique le premier cas du test.
- Nous testons si la variable `ville` est égale à `Angers`.
- Si c'est le cas, nous affichons un message dans la console, puis nous stoppons les tests avec l'instruction `break`, pour sortir du `switch()`.
- Si ce n'est pas le cas, nous effectuons un deuxième test.

- Et ainsi de suite.
- La dernière instruction n'est pas un test, mais indique ce qu'il faut faire par défaut, default:, si aucun test n'est vrai.

Voici l'affichage obtenu :

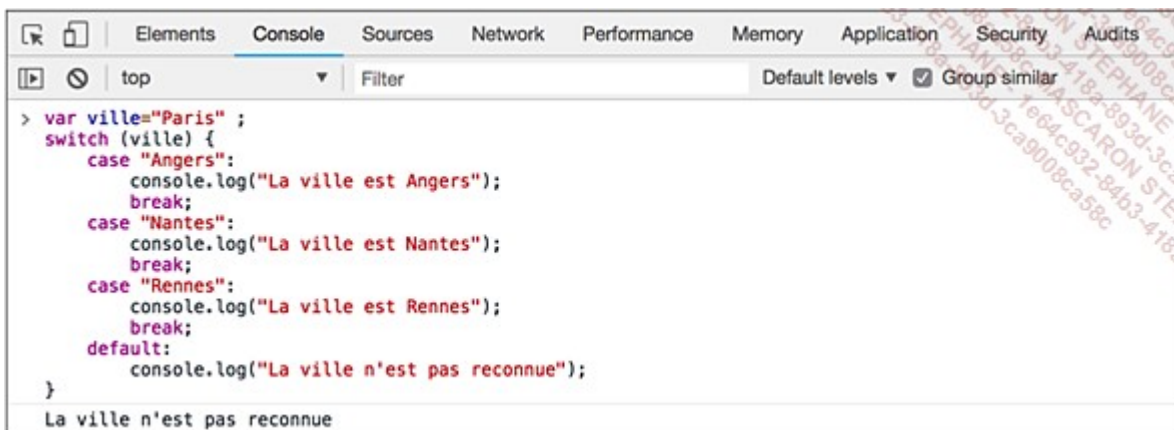


```

> var ville="Nantes" ;
  switch (ville) {
    case "Angers":
      console.log("La ville est Angers");
      break;
    case "Nantes":
      console.log("La ville est Nantes");
      break;
    case "Rennes":
      console.log("La ville est Rennes");
      break;
    default:
      console.log("La ville n'est pas reconnue");
  }
  La ville est Nantes

```

Voici le code utilisé avec une ville non testée, Paris dans cet exemple :



```

> var ville="Paris" ;
  switch (ville) {
    case "Angers":
      console.log("La ville est Angers");
      break;
    case "Nantes":
      console.log("La ville est Nantes");
      break;
    case "Rennes":
      console.log("La ville est Rennes");
      break;
    default:
      console.log("La ville n'est pas reconnue");
  }
  La ville n'est pas reconnue

```

Les boucles

1. Les objectifs

Les boucles permettent d'exécuter du code JavaScript de manière répétitive, tout en contrôlant cette répétition, pour éviter que l'exécution du code voulu ne se répète à l'infini.

Nous allons appréhender la boucle while(), "tant que" et sa variante do while(), ainsi que la boucle for(), "pour".

2. La boucle while()

La boucle while() permet d'exécuter un code tant qu'une condition est vraie. Voici la syntaxe de base de cette boucle :

```

while( condition ){
  Exécuter le code ;
  Evolution de la condition ;
}

```

Étudions cette structure de base :

- Dans les parenthèses de l'instruction `while()`, nous plaçons la condition qui doit être vérifiée pour que la boucle fonctionne. Vous pouvez utiliser tous les opérateurs mathématiques et logiques que nous avons vus précédemment.
- Dans les accolades `{...}`, nous plaçons le code JavaScript qui doit être exécuté tant que la condition est vraie.
- Ensuite, nous devons forcément avoir une évolution de la condition, sinon la boucle s'exécuterait indéfiniment.

Prenons un exemple simple pour comprendre le principe d'une boucle infinie :

```
var a=10 ;
while ( a<100 ){
    console.log( "a est inférieur à 100" ) ;
}
```

Analysons cette structure :

- Nous déclarons une variable numérique `a`, affectée de la valeur 10.
- Nous entrons dans la boucle `while()`.
- Dans la boucle `while()`, nous testons si `a` est inférieur à 100. Ce test renvoie `true`, puisque 10 est effectivement inférieur à 100.
- Puisque le test est valide, nous exécutons le code indiqué entre les accolades `{...}`. Ce code affiche un message dans la console, avec `console.log()`.
- Mais nous sommes toujours dans l'instruction `while()`. Et comme le test conditionnel est toujours vrai, nous affichons à nouveau le message dans la console. Et ainsi de suite, ce message est répété à l'infini, puisque la variable `a` est toujours égale à 10 et qu'elle est donc toujours inférieure à 100.

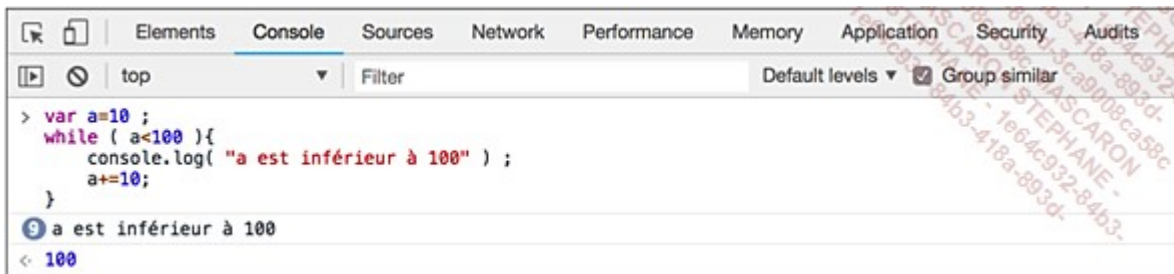
Nous avons bien une boucle infinie, puisque le test conditionnel est et sera toujours vrai. Pour sortir de cette boucle infinie, il faut obligatoirement faire évoluer la condition. Pour cela, nous pouvons faire évoluer la valeur de la variable `a` en l'incrémentant, par exemple.

Voici le code modifié :

```
var a=10 ;
while ( a<100 ){
    console.log( "a est inférieur à 100" ) ;
    a+=10;
}
```

Nous avons ajouté une ligne supplémentaire qui incrémente de 10 la valeur de la variable `a` : `a+=10`. Donc, au bout de 9 passages, la condition n'est plus vraie, la boucle s'arrête et nous sortons de cette boucle.

Voici l'affichage obtenu :

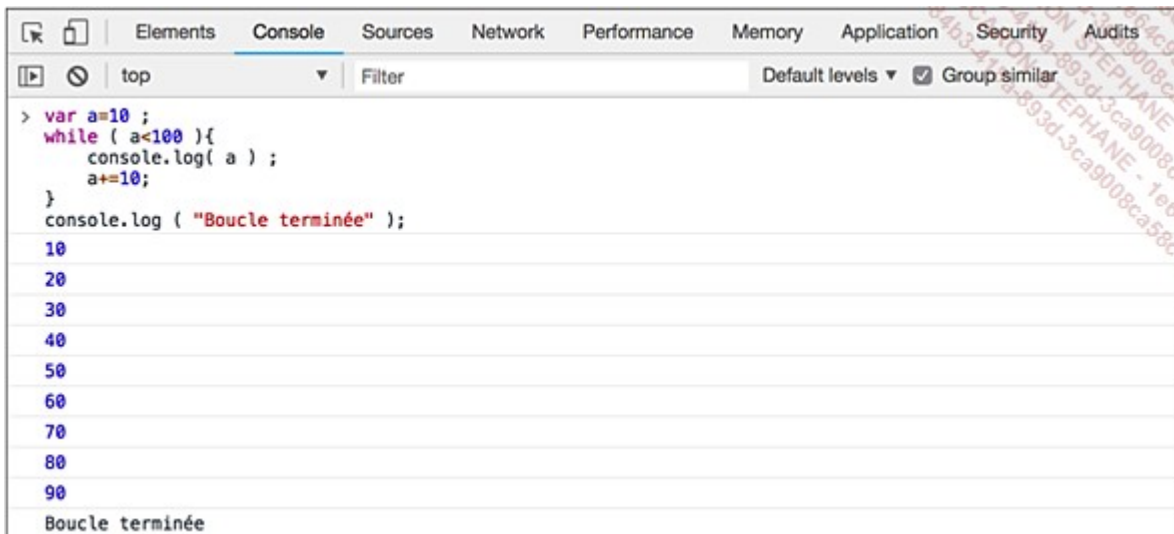


La console nous indique bien qu'il y a eu 9 passages.

Nous pouvons modifier le code pour afficher les valeurs successives de la variable a :

```
var a=10 ;
while ( a<100 ){
    console.log( a ) ;
    a+=10;
}
console.log ( "Boucle terminée" );
```

Voici l'affichage obtenu :



3. La boucle do while()

Voyons maintenant une variante de la boucle "Tant que", c'est la boucle do while(), "Fait tant que". Cette boucle est plus rare, mais vous risquez de la rencontrer. L'objectif est le même que pour la boucle while(), mais, quoiqu'il arrive, l'instruction dans la boucle est exécutée au moins une fois.

Reprenons l'exemple précédent, en l'adaptant à cette variante :

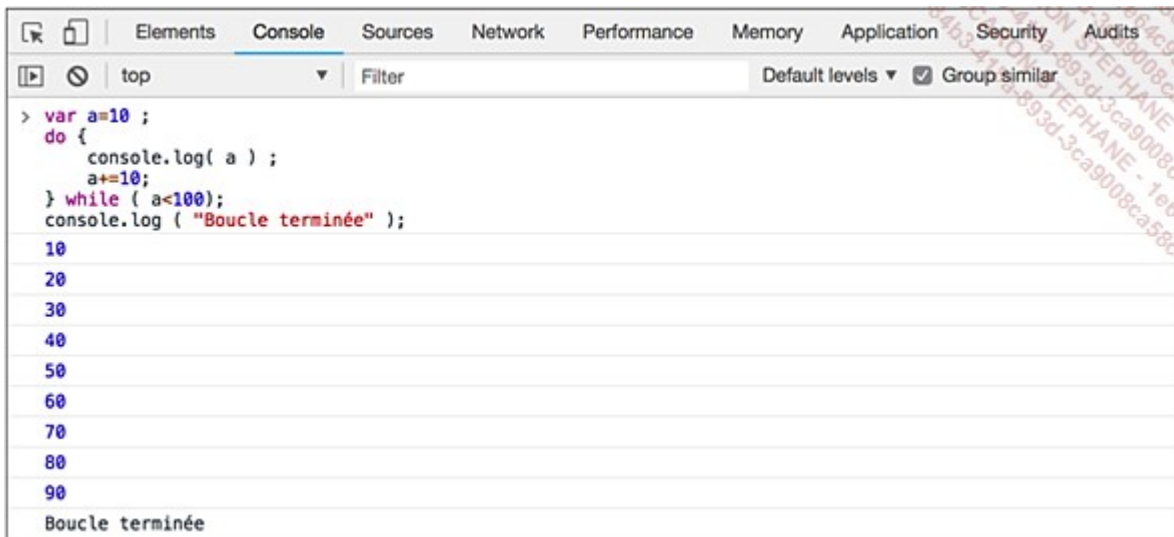
```
var a=10 ;
do {
    console.log( a ) ;
    a+=10;
} while ( a<100);
console.log ( "Boucle terminée" );
```

Analysons cette structure :

- Nous déclarons une variable numérique a, affectée de la valeur 10.

- L'instruction `do{}` exécute ce qui lui est demandé : afficher dans un message de la console la valeur de `a`, soit 10.
- Puis, toujours dans le `do`, nous incrémentons la variable : `a+=10`.
- Et c'est uniquement ensuite que le test conditionnel du `while()` est fait.
- Et du résultat de ce test dépend la poursuite de la boucle (`true`) ou la sortie de cette boucle (`false`).

Voici l'affichage obtenu dans la console :



```

> var a=10 ;
do {
  console.log( a ) ;
  a+=10;
} while ( a<100);
console.log ( "Boucle terminée" );
10
20
30
40
50
60
70
80
90
Boucle terminée

```

4. La boucle `for()`

La boucle `for()`, "pour", permet, elle aussi, d'exécuter un code, en paramétrant l'évolution d'une variable. Nous avons besoin de trois paramètres pour utiliser la boucle `for()` :

- une variable ayant une valeur initiale,
- une condition à tester sur cette variable,
- une évolution de cette variable.

Voici la syntaxe de base de la boucle `for()` :

```

for ( variable initiale ; test conditionnel ; évolution de la variable ) {
  Code à exécuter ;
}

```

Reprenons le même principe que les exemples précédents :

```

for ( var a=10 ; a<100 ; a+=10 ) {
  console.log ( "La variable a = "+a ) ;
}
console.log("Boucle terminée");

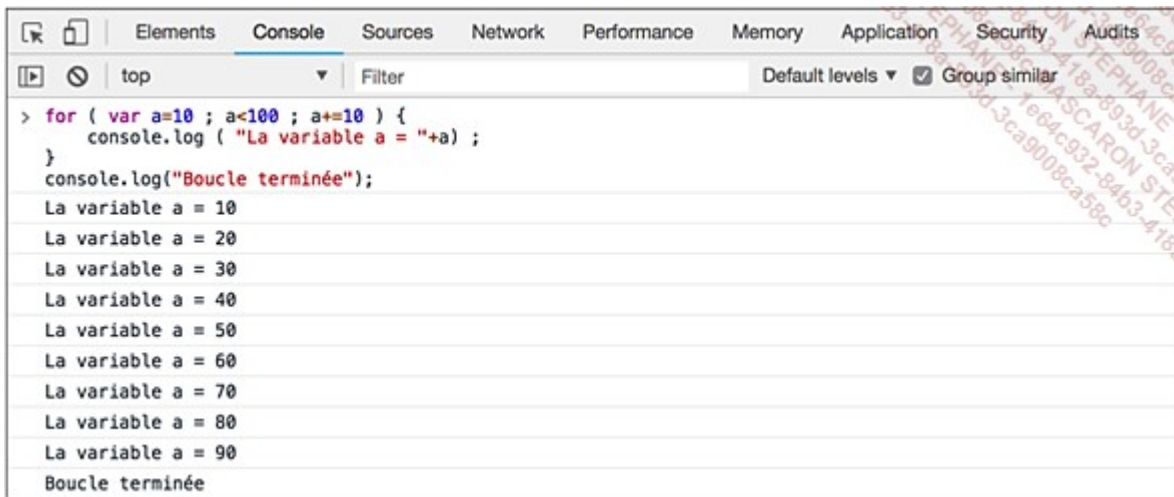
```

Étudions cette structure :

- Nous définissons une boucle `for()`, avec ses trois paramètres.
- Le premier paramètre crée une variable nommée `a` et lui affecte la valeur initiale de 10.

- Le deuxième paramètre teste si la valeur de la variable a est inférieure à 100. Ce qui est vrai pour le premier passage.
- Le troisième paramètre incrémente la variable a de 10, à chaque passage dans la boucle.
- Si le test du deuxième paramètre est vrai, nous exécutons l'instruction qui affiche un message dans la console.
- Quand le test conditionnel n'est plus vrai, nous sortons de la boucle for().

Voici l'affichage obtenu :



```
> for ( var a=10 ; a<100 ; a+=10 ) {
  console.log ( "La variable a = "+a) ;
}
console.log("Boucle terminée");
La variable a = 10
La variable a = 20
La variable a = 30
La variable a = 40
La variable a = 50
La variable a = 60
La variable a = 70
La variable a = 80
La variable a = 90
Boucle terminée
```

5. Les mots-clés de la boucle for()

Dans une boucle for(), vous pouvez effectuer des tests et agir sur l'exécution de cette boucle.

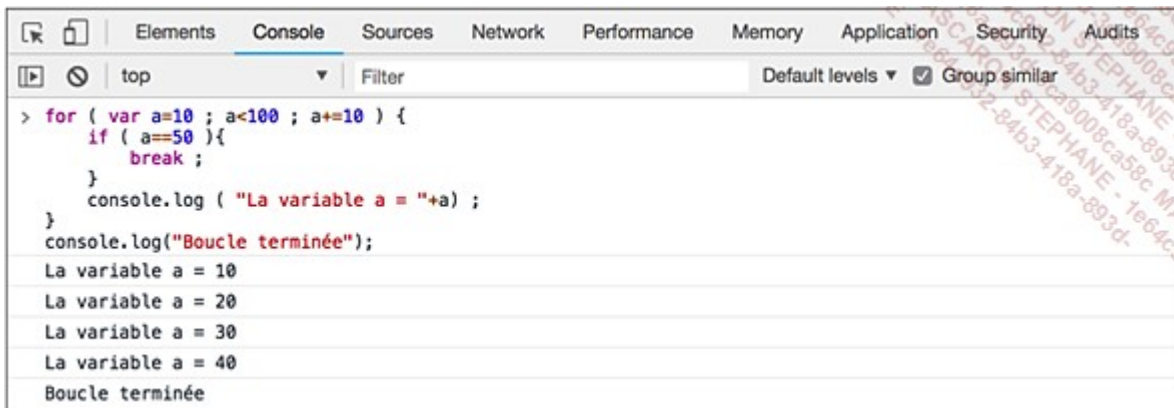
Reprenons l'exemple précédent, en ajoutant le mot-clé break :

```
for ( var a=10 ; a<100 ; a+=10 ) {
  if ( a==50 ){
    break ;
  }
  console.log ( "La variable a = "+a) ;
}
console.log("Boucle terminée");
```

Analysons nos modifications :

- Nous avons ajouté un test if() pour tester si la variable a est égale à 50.
- Si c'est le cas, nous demandons à stopper la boucle et à en sortir, avec le mot-clé break.

Voici l'affichage obtenu :



```
> for ( var a=10 ; a<100 ; a+=10 ) {  
  if ( a==50 ){  
    break ;  
  }  
  console.log ( "La variable a = "+a) ;  
}  
console.log("Boucle terminée");  
La variable a = 10  
La variable a = 20  
La variable a = 30  
La variable a = 40  
Boucle terminée
```

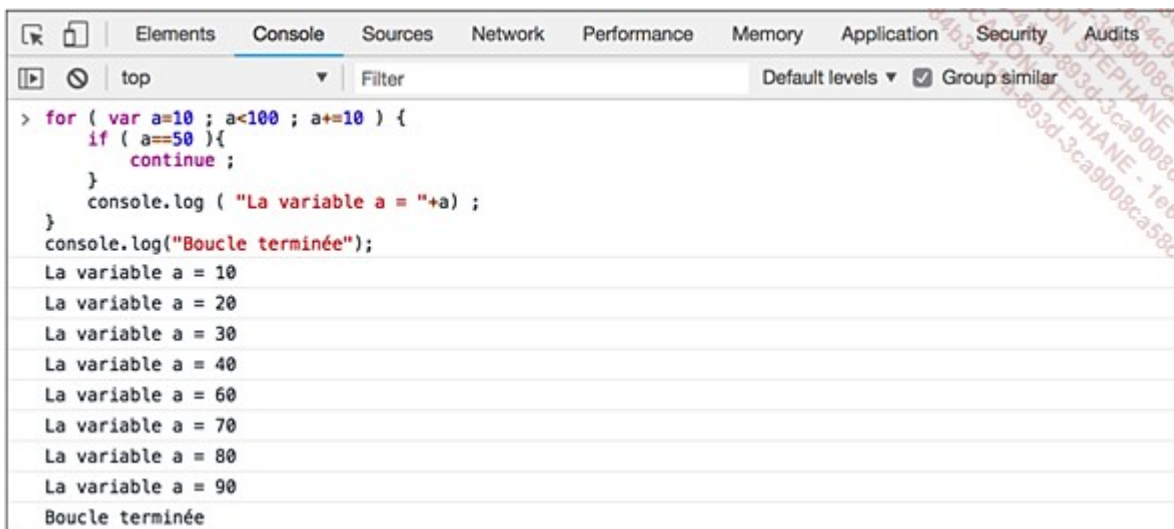
Nous arrêtons bien la boucle for() lorsque la variable a est égale à 50. La dernière valeur affichée dans la console est bien 40.

Deuxième mot-clé que nous pouvons utiliser dans la boucle for(), c'est continue. À nouveau, nous allons pouvoir faire un test qui va ignorer l'exécution du code si le test renvoie true, mais la boucle ne sera pas interrompue, elle continuera.

Reprenons l'exemple précédent, en ajoutant le mot-clé continue :

```
for ( var a=10 ; a<100 ; a+=10 ) {  
  if ( a==50 ){  
    continue ;  
  }  
  console.log ( "La variable a = "+a) ;  
}  
console.log("Boucle terminée");
```

Voici l'affichage obtenu :



```
> for ( var a=10 ; a<100 ; a+=10 ) {  
  if ( a==50 ){  
    continue ;  
  }  
  console.log ( "La variable a = "+a) ;  
}  
console.log("Boucle terminée");  
La variable a = 10  
La variable a = 20  
La variable a = 30  
La variable a = 40  
La variable a = 60  
La variable a = 70  
La variable a = 80  
La variable a = 90  
Boucle terminée
```

Vous voyez bien que seule la valeur 50 n'est pas affichée, puisqu'elle renvoie vrai au test du if(). Par contre, toutes les autres valeurs sont bien affichées, grâce au mot-clé continue.

Les fonctions

1. Les objectifs

Les fonctions vont vous permettre de créer des lignes de code que vous allez regrouper en un bloc que vous nommerez. Cela va permettre de mieux structurer votre code, de spécialiser certaines fonctionnalités et d'appeler chaque fonction plusieurs fois, dans des scripts et des pages différentes.

2. Créer une fonction

Voilà la syntaxe de base de création d'une fonction :

```
function ma_fonction () {  
    instruction1 ;  
    instruction2 ;  
    ...  
}  
ma_fonction ();
```

Détaillons ce code :

- Nous utilisons le mot-clé réservé function pour créer une nouvelle fonction.
- Cette fonction est nommée ma_fonction.



Notez bien que les fonctions sont des variables, donc vous devez respecter les règles de nommage que nous avons vues précédemment.

- Le nom de la fonction est obligatoirement suivi par des parenthèses, qui permettront de passer d'éventuels paramètres à cette fonction.
- Nous avons ensuite une paire d'accolades.
- Entre ces accolades, nous indiquons toutes les instructions que doit exécuter cette fonction.
- La dernière ligne appelle la fonction afin qu'elle soit exécutée.

Nous allons créer une fonction simple afin d'en comprendre l'utilisation. Cette fonction sera créée dans un fichier JavaScript qui sera nommé dans cet exemple **ma-fonction.js**.

Voici le code de cette fonction :

```
var a=12, b=23 ;  
function ma_fonction(){  
    var resultat_addition = a + b ;  
    document.getElementById("resultat").innerHTML=resultat_addition;  
}  
ma_fonction();
```

Détaillons le code de ce fichier :

- Nous créons deux variables numériques, a et b, affectées de deux valeurs.

- La fonction est nommée `ma_fonction()`.
- La première ligne d'instruction crée une nouvelle variable `resultat_addition` qui va calculer l'addition des deux variables précédentes.
- La deuxième ligne d'instruction va rechercher dans le document HTML, `document`, un élément, `getElementById()`, ayant l'identifiant `resultat`.
- Dans cet élément, `document.getElementById("resultat")`, la fonction va afficher, `innerHTML`, la valeur de la variable `resultat_addition`.
- La dernière ligne de ce script est tout simplement l'appel de cette fonction, `ma_fonction()`, afin qu'elle soit exécutée.

Voici maintenant la structure du document HTML :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p>Le résultat du calcul est : <strong id="resultat"></strong>.</p>
    <script src="ma-fonction.js"></script>
  </body>
</html>
```

Dans l'élément `<p>`, nous avons bien un élément HTML, `` dans cet exemple, qui possède l'identifiant `resultat`.

Donc, au chargement de cette page, nous avons cet affichage :

Le résultat du calcul est : 35.

L'exemple à télécharger est dans le dossier **Chapitre-02-I/Exemple-01**.

3. Utiliser des paramètres

Nous avons vu précédemment que les parenthèses d'une fonction permettent de passer des paramètres dans cette fonction. Dans cet exemple, nous allons exploiter cette fonctionnalité en appelant plusieurs fois cette fonction, en lui passant des paramètres différents.

Voici la structure modifiée du fichier HTML :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <p>Le résultat 1 du calcul est : <strong id="resultat_un"></strong>.</p>
    <p>Le résultat 2 du calcul est : <strong id="resultat_deux"></strong>.</p>
    <script src="ma-fonction.js"></script>
  </body>
```

</html>

Nous avons maintenant deux éléments <p>, qui contiennent deux éléments avec deux identifiants différents : id="resultat_un" et id="resultat_deux".

Voici le code JavaScript modifié :

```
function ma_fonction(nb1,nb2,identifiant){
    var resultat_addition = nb1 + nb2 ;
    document.getElementById(identifiant).innerHTML=resultat_addition;
}
ma_fonction(12,23,"resultat_un");
ma_fonction(34,45,"resultat_deux");
```

Détaillons le code :

- La fonction ma_fonction() utilise trois paramètres : nb1, nb2 et identifiant.
- Nous avons toujours la même variable de calcul, resultat_addition, qui va cette fois utiliser les paramètres de la fonction, nb1 et nb2, pour effectuer ce calcul.
- La fonction va rechercher l'élément HTML dont l'identifiant sera la valeur du paramètre identifiant.
- Ensuite, le premier appel de la fonction passe trois valeurs pour les trois paramètres :
 - 12 pour nb1,
 - 23 pour nb2,
 - "resultat_un" pour identifiant.
- Pour terminer, le deuxième appel de la fonction passe trois autres valeurs pour les trois paramètres :
 - 34 pour nb1,
 - 45 pour nb2,
 - "resultat_deux" pour identifiant.

Voici l'affichage obtenu :

Le résultat 1 du calcul est : 35.

Le résultat 2 du calcul est : 79.

L'exemple à télécharger est dans le dossier **Chapitre-02-I/Exemple-02**.

4. Renvoyer les résultats

Si nous pouvons envoyer des paramètres à une fonction, nous pouvons aussi demander à cette fonction de nous renvoyer des résultats.

Pour cet exemple, la page HTML (**exemple-03.html**) fait juste appel au fichier JavaScript :

```
<!DOCTYPE html>
<html>
```

```
<head>
  <meta charset="UTF-8">
  <title>Mon script</title>
</head>
<body>
  <script src="ma-fonction.js"></script>
</body>
</html>
```

Voici le contenu du fichier **ma-fonction.js** :

```
function ma_fonction(nb1,nb2){
  var resultat_addition = nb1 + nb2 ;
  return resultat_addition ;
} var calcul = ma_fonction(12,23);
alert("Le résultat est "+calcul);
```

Analysons ce code :

- La fonction est identique à la précédente, mais avec un paramètre de moins. Nous n'utilisons que deux variables, nb1 et nb2, pour le calcul.
- L'instruction de calcul est la même, avec la variable resultat_addition.
- La deuxième instruction de la fonction utilise le mot-clé return qui permet d'indiquer à la fonction que nous souhaitons qu'elle renvoie le résultat du calcul, c'est-à-dire la valeur de la variable resultat_addition, à l'élément qui va appeler cette fonction.
- Ensuite, nous créons une nouvelle variable nommée calcul qui va stocker le résultat de la fonction ma_fonction(). Nous passons à cette fonction les deux paramètres demandés, 12 et 23 dans cet exemple. Donc la variable calcul fait appel à la fonction ma_fonction(), qui lui renvoie son résultat.
- Puis nous affichons le résultat renvoyé par la fonction et stocké dans la variable, dans une fenêtre d'alerte.

Voici l'affichage obtenu :



L'exemple à télécharger est dans le dossier **Chapitre-02-I/Exemple-03**.



Notez que si vous indiquez trop de paramètres, cela n'est pas gênant, ils seront ignorés. Par contre, si vous ne passez pas assez de paramètres, vous aurez un message d'erreur.

Voici l'affichage obtenu lorsque des paramètres manquent :



Cet affichage est normal car la fonction attend deux variables pour faire son calcul. Si elle n'en a qu'une seule, la deuxième variable est alors undefined et JavaScript ne peut pas faire de calcul avec ce type de données. Donc le script renvoie le type NaN, **Not a Number**.

La portée des variables

1. Créer une variable locale

Abordons maintenant un point très important : la portée des variables. La portée des variables indique où nous pouvons utiliser les variables que nous créons.

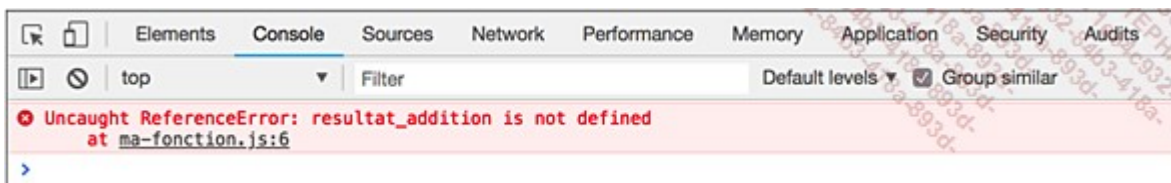
Reprenons l'exemple précédent en nous disant que, puisque nous avons défini la variable resultat_addition, pourquoi ne pas l'utiliser dans l'affichage du résultat ?

Voici le code du fichier **ma-fonction.js** modifié :

```
function ma_fonction(nb1,nb2){  
    var resultat_addition = nb1 + nb2 ;  
    return resultat_addition ;  
}  
var calcul = ma_fonction(12,23);  
alert("Le résultat est "+resultat_addition);
```

Dans l'instruction alert, nous utilisons directement la variable resultat_addition.

Le fichier **exemple-01.html** est inchangé. Voici l'affichage obtenu dans la console :



La console nous affiche ce message : **resultat_addition is not defined**. Ce message indique que la variable resultat_addition n'est pas définie, elle est inconnue pour le script.

C'est un affichage parfaitement normal. En effet, la variable resultat_addition a été définie dans la fonction ma_fonction(), entre ses accolades et surtout avec le mot-clé var. De ce fait même, la portée de cette variable est limitée au bloc de code où elle a été créée, c'est-à-dire dans la fonction. En dehors de cette fonction, cette variable est totalement inconnue pour les autres instructions.



La portée d'une variable créée avec le mot-clé var dans une fonction est limitée à cette fonction. C'est une variable locale.

L'exemple à télécharger est dans le dossier **Chapitre-02-J/Exemple-01**.

2. Créer une variable globale dans une fonction

Si nous souhaitons utiliser partout dans le script une variable définie dans une fonction, il ne faut pas utiliser le mot-clé `var`. Dans ce cas, la variable possède une portée globale. C'est-à-dire que cette variable globale est utilisable partout dans le script.

Voici un exemple qui reprend la même structure que précédemment :

```
function ma_fonction(nb1,nb2){
    resultat_addition = nb1 + nb2 ;
    return resultat_addition ;
}
var calcul = ma_fonction(12,23);
alert("Le résultat est "+resultat_addition);
```

La seule différence est que la variable `resultat_addition`, qui est toujours créée dans la fonction `ma_fonction()`, est déclarée sans l'utilisation du mot-clé `var`.

Voici l'affichage obtenu :



L'exemple à télécharger est dans le dossier **Chapitre-02-J/Exemple-02**.

3. Créer une variable globale dans le script

Si nous voulons utiliser des variables partout dans le fichier du script avec l'utilisation du mot-clé `var`, il faut les créer en dehors de toute fonction. Elles sont alors de type variable globale, en opposition aux variables locales qui sont définies dans une fonction.

Voilà le script modifié pour avoir la variable globale :

```
var resultat_addition ;
function ma_fonction(nb1,nb2){
    resultat_addition = nb1 + nb2 ;
    return resultat_addition ;
}
var calcul = ma_fonction(12,23);
alert("Le résultat est "+resultat_addition);
```

- La première ligne du script définit la variable `resultat_addition`, avec le mot-clé `var`.



Notez que nous ne donnons pas de valeur initiale à cette variable. Cela n'est absolument pas gênant.

- Dans la fonction `ma_fonction()`, il ne faut pas saisir le mot-clé `var` pour utiliser la variable, puisqu'elle a été déjà créée. Ici, c'est juste une affectation par l'intermédiaire d'un calcul.

- Enfin, dans la dernière ligne, nous pouvons sans problème utiliser la variable globale resultat_addition.

L'affichage obtenu est correctement le même que précédemment :



L'exemple à télécharger est dans le dossier **Chapitre-02-J/Exemple-03**.

Les objets du code JavaScript

1. Définir la notion d'objet

En programmation, un "objet" est une notion fondamentale de tous les langages modernes, ils sont très largement utilisés. En programmation, la définition d'un objet peut être celle-ci :



Un objet est une collection de propriétés et de méthodes.

- Une **propriété** est une caractéristique d'un objet.
- Une **méthode** est une fonctionnalité d'un objet.

Prenons une analogie simple : l'être humain est un objet ! Il est bien la collection de très nombreuses propriétés et de non moins nombreuses méthodes.

Pour les humains, nous pouvons avoir comme propriétés la taille, la couleur des cheveux, le sexe, le poids, le nom... Vous avez noté que ces exemples de propriétés sont tous des paramètres mesurables avec des valeurs. Comme en programmation, les propriétés sont des variables de type numérique, chaîne de caractères...

Pour les humains, nous pouvons avoir comme méthodes marcher, manger, dormir, penser... Vous avez noté que les exemples de méthodes sont tous des verbes d'action. Comme en programmation, les méthodes sont des fonctions qui exécutent des instructions.

Sachez qu'en JavaScript, presque tous les éléments que nous utilisons sont des objets. Les variables sont des objets, les fonctions sont des objets, les chaînes de caractères sont des objets, tout comme les dates et les tableaux, que nous allons aborder. Si vous souhaitez avoir la liste de tous les objets natifs du JavaScript, allez à cette URL :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux, sur le site des développeurs de Mozilla.



Notez bien que vous pouvez vous-même créer les objets nécessaires à vos codes JavaScript.

2. Créer un nouvel objet

Nous allons prendre tout de suite un exemple concret pour étudier la création des objets. Reprenons l'exemple de l'être humain, qui est un objet que nous pouvons qualifier de générique. C'est l'objet générique de base **Humain** qui est à l'origine de tous les humains. Donc si nous voulons créer un nouvel **humain**, nous allons nous baser sur l'objet générique de base **Humain**.

Pour créer des objets, le langage JavaScript nous propose deux méthodes :

- La première méthode est littérale. Elle permet de créer un nouvel objet en créant une nouvelle variable et en indiquant les propriétés que nous souhaitons y stocker.

- La deuxième méthode utilise le constructeur d'objet générique Object().

3. La méthode de création littérale

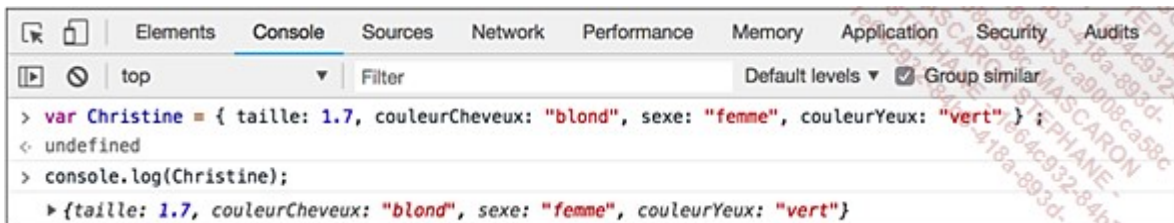
Nous allons créer un nouvel "humain" en lui attribuant les propriétés voulues. Voici l'exemple :

```
var Christine = { taille: 1.7, couleurCheveux: "blond", sexe: "femme",  
couleurYeux: "vert" } ;
```

Analysons la syntaxe :

- Nous déclarons une variable, Christine, avec le mot-clé var.
- Nous attribuons des valeurs avec l'opérateur =.
- Toutes les valeurs sont placées entre les deux accolades { }.
- Chaque propriété est constituée d'un couple : nom de la propriété et valeur de la propriété.

Si nous créons ce nouvel objet dans la console JavaScript d'un navigateur et qu'ensuite nous affichons ses données avec console.log(), nous vérifions bien la création de ce nouvel objet :



Dans la console, vous notez la présence d'un triangle devant la liste des propriétés indiquées entre les deux accolades. Si vous cliquez sur ce triangle, vous affichez chaque propriété et sa valeur dans une seule ligne.



Notez que si, au début de votre programme, vous ne connaissez pas encore les propriétés à attribuer à un objet, vous pouvez parfaitement créer un objet vide et renseigner par la suite ses propriétés.

```
var Mathis = {} ;  
...  
Mathis.taille = 1.86 ;  
...  
Mathis.sexe = "homme" ;
```

Pour attribuer des propriétés plus tard dans le script, nous utilisons cette syntaxe :

- Créez la variable, var Mathis, et ne lui affectez, =, aucune propriété entre les deux accolades {}.

- Continuez votre script.
- Puis, saisissez le nom de l'objet, Mathis dans cet exemple.
- Saisissez ensuite le caractère point ., qui permet de faire le lien entre l'objet et la propriété à définir.
- Saisissez le nom de la propriété, taille dans cet exemple.
- Utilisez l'opérateur d'affectation =.
- Indiquez la valeur numérique ou la chaîne de caractères de cette propriété, 1.86 dans cet exemple.

Utilisons la console JavaScript d'un navigateur pour vérifier la création de ce nouvel objet :

```

> var Mathis = {};
< undefined
> Mathis.taille = 1.86;
< 1.86
> Mathis.sexe = "homme";
< "homme"
> console.log(Mathis);
▼ {taille: 1.86, sexe: "homme"}
  sexe: "homme"
  taille: 1.86
  __proto__: Object

```

4. La méthode de création avec un constructeur

La deuxième méthode pour créer un objet consiste à utiliser le constructeur générique Object().

Voici sa syntaxe :

```

var Celia = new Object() ;
Celia.sexe = "femme" ;
Celia.taille = 1.7 ;
Celia.couleurCheveux = "chatain" ;

```

Analysons la syntaxe :

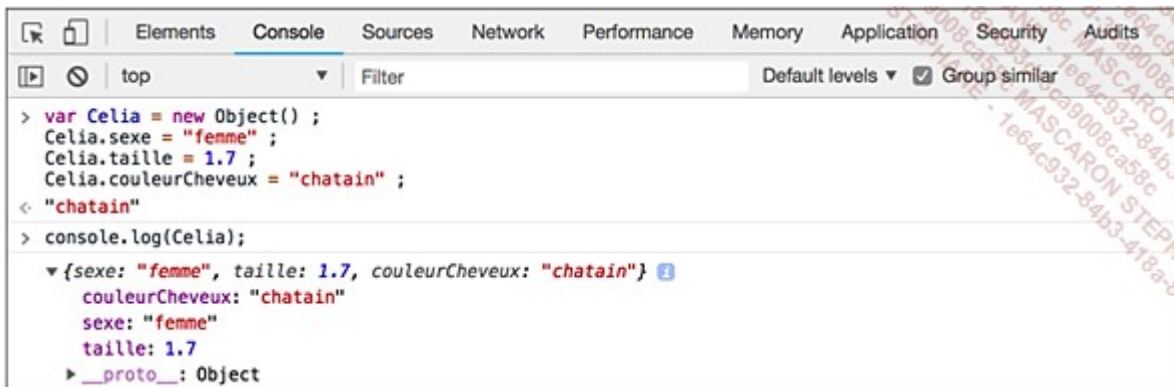
- Nous déclarons une variable avec le mot-clé var.
- Nous déclarons ensuite le nom de cette variable, de ce futur nouvel objet, Celia dans cet exemple.
- Nous indiquons que cette variable est une nouvelle entité avec l'opérateur =.
- Nous utilisons le constructeur new pour créer une nouvelle entité.
- Cette nouvelle entité est un objet qui utilise l'objet générique Object().



Notez bien que le mot **Object** qui commence par un O majuscule qui est obligatoire et suivi par un couple de parenthèses.

- Ensuite, pour ce nouvel objet, nous indiquons les propriétés avec les couples nom/valeur.

Comme précédemment, nous pouvons vérifier tout cela dans la console JavaScript d'un navigateur :



```

> var Celia = new Object() ;
  Celia.sexe = "femme" ;
  Celia.taille = 1.7 ;
  Celia.couleurCheveux = "chatain" ;
< "chatain"
> console.log(Celia);
▼ {sexe: "femme", taille: 1.7, couleurCheveux: "chatain"} ⓘ
  couleurCheveux: "chatain"
  sexe: "femme"
  taille: 1.7
  ► __proto__: Object

```



Notez que cette deuxième méthode est un peu plus verbeuse, moins lisible et moins efficace pour les navigateurs.

5. Récupérer les valeurs des propriétés

Naturellement, l'objectif suivant va être de récupérer la valeur d'une des propriétés renseignées lors de la création de l'objet. Pour ce faire, JavaScript nous propose deux syntaxes.

Voici la création du nouvel objet utilisé dans ces exemples :

```
var Christine = { taille: 1.7, couleurCheveux: "blond",
sexe: "femme", couleurYeux: "vert" } ;
```

La première méthode, la notation à point, indique le nom de l'objet, suivi par un point, et se termine par le nom de la propriété dont nous souhaitons récupérer la valeur.

Voici la syntaxe pour récupérer la valeur de la propriété taille de l'objet Christine :

```
Christine.taille ;
```

Dans une nouvelle variable, nous indiquons cette propriété que nous affichons avec l'instruction console.log() :



```

> var Christine = { taille: 1.7, couleurCheveux: "blond", sexe: "femme", couleurYeux: "vert" } ;
< undefined
> Christine.taille ;
< 1.7

```

Nous pouvons aussi utiliser l'instruction console.log() :



```

> var Christine = { taille: 1.7, couleurCheveux: "blond", sexe: "femme", couleurYeux: "vert" } ;
< undefined
> console.log(Christine.taille);
1.7

```

La deuxième méthode, la notation avec les crochets, repose sur les mêmes principes que celle à point, à la différence près que la propriété est indiquée entre crochets et entre guillemets. Voici un exemple :

```
Christine["taille"] ;
```

L'intérêt de cette syntaxe est d'éviter d'obtenir des erreurs de rendu du code, si le nom d'une propriété comporte des caractères interdits par JavaScript.

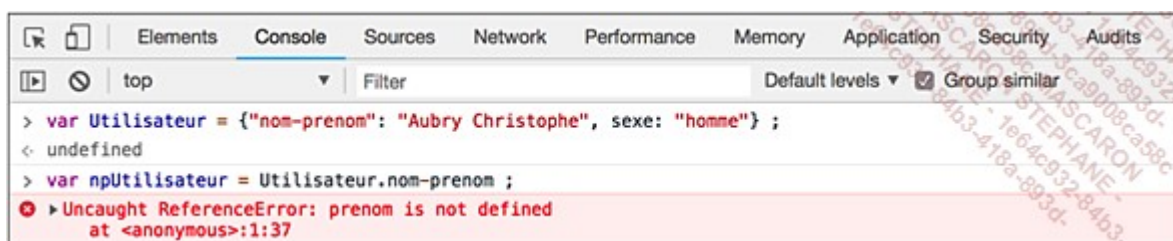
Voici un exemple précis. Nous créons un nouvel objet nommé Utilisateur avec, entre autres, une propriété nommée nom-prenom.

```
var Utilisateur = {"nom-prenom": "Aubry Christophe", sexe: "homme"} ;
```

Nous ne pouvons pas utiliser la syntaxe à point pour récupérer la valeur de la propriété "nom-prenom".

```
var npUtilisateur = Utilisateur.nom-prenom ;
```

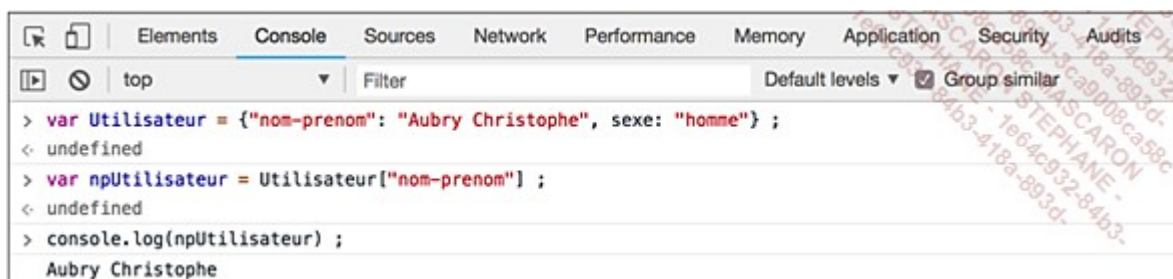
JavaScript va interpréter le tiret - entre les mots nom et prenom comme un opérateur de soustraction. Nous aurons donc une erreur :



Par contre, avec la syntaxe en crochets :

```
var npUtilisateur = Utilisateur["nom-prenom"] ;
```

Nous n'aurons aucune erreur :



6. Les méthodes des objets

Nous l'avons défini au début de ce titre, les objets sont des collections de propriétés et de méthodes. Nous allons maintenant aborder les méthodes des objets.

Une méthode est simplement une propriété qui a non pas une valeur, mais qui utilise une fonction.

Les méthodes s'écrivent avec un nom suivi d'un couple de parenthèses : ma_methode(). Les parenthèses, obligatoires, servent à passer d'éventuels paramètres à la méthode.

Dans un fichier HTML, nous allons reprendre nos objets humains, avec de nouvelles propriétés :

```

<script>
  var Personne_1 = {
    prenom: "Christine",
    nom: "Duteil",
    sexe: "femme",
    couleurYeux: "vert",
  };
</script>

```

Nous allons ajouter une méthode qui va simplement reprendre les valeurs du prénom et du nom pour afficher un bonjour personnalisé.

Voici l'objet modifié :

```

<script>
  var Personne_1 = {
    prenom: "Christine",
    nom: "Duteil",
    sexe: "femme",
    couleurYeux: "vert",
    bonjour : function(){
      alert ( "Bonjour de " + Personne_1.prenom + " " +
Personne_1.nom ) ;
    }
  };
  Personne_1.bonjour();
</script>

```

Voici l'affichage obtenu :



Étudions le code au niveau de la méthode :

- Une méthode est juste une propriété qui utilise une fonction. Donc notre méthode est déclarée avec un nom, bonjour et le mot-clé function().
- Entre les accolades de la fonction, nous avons le code qui doit être exécuté. Dans cet exemple, c'est une simple alerte qui récupère le prénom et le nom, avec une concaténation de textes fixes.
- Ensuite, après la création de l'objet, nous avons l'appel à cette méthode :
Personne_1.bonjour();

L'exemple à télécharger est dans le dossier **Chapitre-02-J/exemple-01.html**.

Nous pouvons améliorer le code en évitant de saisir le nom de l'objet dans la fonction. Nous remplaçons Personne_1 par le mot-clé this. En effet, il ne sert à rien de préciser le nom de l'objet, puisque la déclaration de la fonction se fait dans l'objet. L'autre avantage d'utiliser this est que c'est un mot-clé générique qui va fonctionner partout, dans toutes les fonctions des objets.

Voici le script modifié :

```

<script>
    var Personne_1 = {
        prenom: "Christine",
        nom: "Duteil",
        sexe: "femme",
        couleurYeux: "vert",
        bonjour : function(){
            alert ( "Bonjour de " + this.prenom + " " + this.nom ) ;
        }
    } ;
    Personne_1.bonjour();
</script>

```

L’affichage obtenu est, bien sûr, strictement identique.

L’exemple à télécharger est dans le dossier **Chapitre-02-J/exemple-02.html**.

Les tableaux

1. Utiliser les tableaux

Nous avons vu dans une section précédente que les variables permettent de stocker une valeur, mais pas plus. Si vous avez besoin de stocker plusieurs variables dans un même élément, vous devez utiliser un tableau. Dès lors que vous avez besoin de gérer des listes de variables, vous pouvez utiliser des tableaux. C’est le même principe dans la vie de tous les jours : liste de courses, liste des anniversaires, liste de tâches à faire, liste des livres à lire ou des disques à écouter... Et bien sûr, les tableaux sont des objets !



Notez bien que les tableaux peuvent contenir n’importe quel type de données : variables numériques, chaîne de caractères, mais aussi des fonctions et des tableaux... Les tableaux sont des conteneurs à plusieurs dimensions.

2. Créer un tableau

Nous pouvons créer un tableau avec le constructeur new appliqué sur l’objet Array().

```
var mesDonnees = new Array("Paul", "Valérie", "Pierre", "Julie") ;
```

Et vous pouvez aussi utiliser la syntaxe littérale :

```
var mesDonnees = ["Paul", "Valérie", "Pierre", "Julie"] ;
```

Ce sont les crochets qui indiquent au JavaScript qu’il a affaire à un tableau.

C’est assez usuellement cette dernière méthode qui est la plus utilisée.

Vous pouvez aussi créer un tableau vide et le remplir par la suite :

```

var mesDonnees = [] ;
...
mesDonnees[0] = "Paul" ;
mesDonnees[1] = "Valérie" ;
...

```



```
mesDonnees[2] = "Pierre" ;  
...  
mesDonnees[3] = "Julie" ;
```

Il est important de bien noter la syntaxe d'ajout de données dans un tableau :

- Nous commençons par créer un tableau : `var mesDonnees`.
- Ce tableau est affecté d'un contenu, avec l'opérateur `=`.
- Ce tableau est vide de toute donnée, mais il y a quand même les deux crochets `[]`.

Ensuite, plus loin dans le code, nous pouvons alimenter le tableau :

- Nous utilisons le nom du tableau, `mesDonnees`.
- Immédiatement après, sans espace, nous avons une paire de crochets `[]`.
- Entre les crochets, nous avons la valeur d'index du tableau. Chaque valeur d'index indique le numéro de position de l'entrée dans le tableau.



Attention, notez-le bien, la première position est toujours **0**. Mais vous n'êtes pas obligé de commencer à 0.

- Ensuite, nous avons l'opérateur d'affectation `=`, suivi par la valeur voulue pour cette entrée, des chaînes de caractères dans cet exemple.

3. Récupérer une valeur d'un tableau

Pour récupérer une valeur d'un tableau, nous devons utiliser la syntaxe en crochets et indiquer le numéro d'index. Souvenez-vous que la première entrée a pour index **0**.

Voici l'exemple d'un code dans un fichier HTML :

```
<script>  
  var mesDonnees = ["Paul", "Valérie", "Pierre", "Julie"] ;  
  alert ("La deuxième entrée est " + mesDonnees[1] + ".");  
</script>
```

Voici l'affichage obtenu :



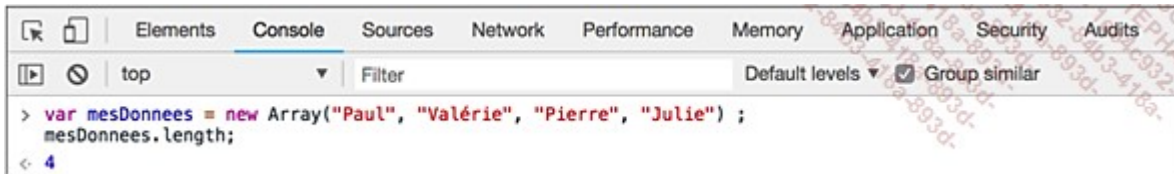
4. Une propriété et une méthode des tableaux

La propriété `length` des tableaux est très utilisée en JavaScript. Elle permet de connaître le nombre d'entrées présentes dans un tableau.

Pour l'exploiter, nous utilisons la syntaxe à point : `nom_du_tableau.length` :

```
var mesDonnees = new Array("Paul", "Valérie", "Pierre", "Julie") ;  
mesDonnees.length;
```

Voici l'affichage obtenu dans la console :

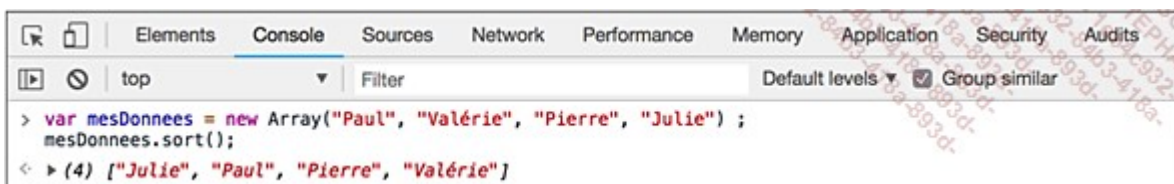


Dans l'exemple de ce tableau, nous avons bien 4 entrées : **Paul**, **Valérie**, **Pierre** et **Julie**.

Voyons un exemple d'une méthode. La méthode `sort` permet de trier par ordre croissant les données du tableau.

```
var mesDonnees = new Array("Paul", "Valérie", "Pierre", "Julie") ;  
mesDonnees.sort();
```

Voici l'affichage obtenu dans la console :



5. Les propriétés et les méthodes de l'objet Array()

L'objectif de cet ouvrage n'est pas de lister toutes les propriétés et toutes les méthodes de tous les objets du JavaScript. Vous avez dans les livres des Éditions ENI des ouvrages de référence sur le sujet.

Voici la page du site des développeurs de Mozilla consacrée à l'objet `Array()` :

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

Les chaînes de caractères

1. Créer un objet de type chaîne de caractères

Les chaînes de caractères sont des objets à part entière. Nous pourrions donc en exploiter des propriétés et des méthodes.

En JavaScript, les objets de texte, de type chaîne de caractères, sont de type `String()`. Pour créer un objet de type chaîne de caractères, nous devons utiliser la classique syntaxe avec `var` et `=` :

```
var unePhrase = "Voici une phrase" ;
```

Souvenez-vous qu'il est possible d'utiliser le caractère guillemet double `"` ou simple `'` pour délimiter la chaîne de caractères. Alors pourquoi deux caractères sont-ils utilisables ? Simplement pour avoir la possibilité d'insérer dans le texte des guillemets, par exemple :

```
var unePhrase = 'Il a dit "Voici une phrase".' ;
```

Attention, si vous voulez insérer une apostrophe dans une phrase, vous devez utiliser le caractère d'échappement pour qu'elle soit interprétée comme du texte et non pas comme du code :

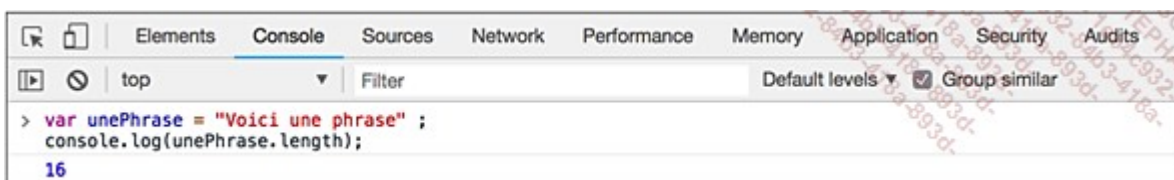
```
var unePhrase = 'Il a dit C\'est une phrase.' ;
```

2. Une propriété et une méthode des chaînes de caractères

Comme pour les tableaux, nous pouvons utiliser la propriété `length` pour connaître le nombre de tous les caractères, espaces et signes de ponctuation compris :

```
var unePhrase = "Voici une phrase" ;  
console.log(unePhrase.length);
```

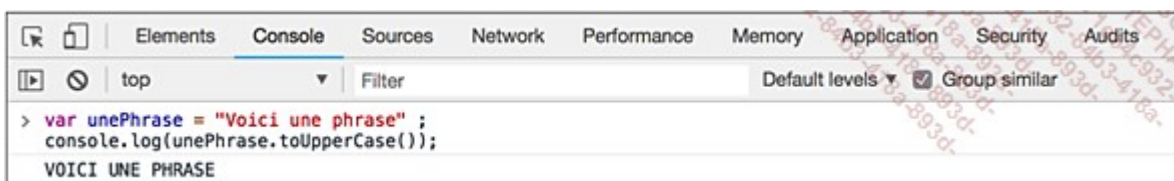
Voici l'affichage obtenu dans la console :



Pour mettre le texte tout en majuscules, nous pouvons utiliser la méthode `toUpperCase`.

```
var unePhrase = "Voici une phrase" ;  
console.log(unePhrase.toUpperCase());
```

Voici l'affichage obtenu dans la console :



3. Les propriétés et les méthodes de l'objet String()

Voici la page du site des développeurs de Mozilla consacrée à l'objet String() :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/String. Vous y trouverez toutes les propriétés et toutes les méthodes de l'objet String().

Les dates

1. Créer des dates

Les dates sont très présentes dans les sites web : dates de création d'un article, date de modification, date d'un commentaire, date de réservation...

Pour créer une date, nous pouvons utiliser la syntaxe classique avec le constructeur appliqué à l'objet Date() :

```
var aujourd'hui = new Date() ;
```

Et nous pouvons l'afficher dans la console :

```
var aujourd'hui = new Date() ;  
console.log( aujourd'hui );
```

Voici l'affichage obtenu :

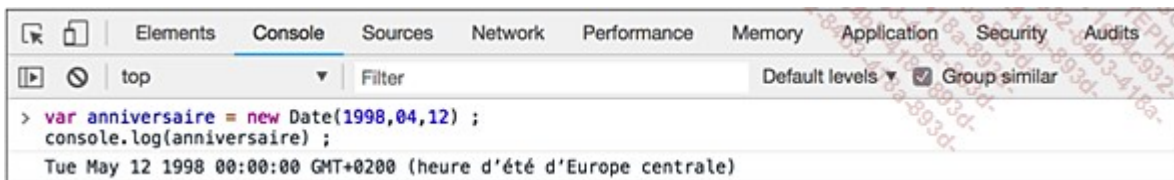


Le résultat affiché est très précis : nous avons la date, l'heure, le fuseau horaire GMT et l'indication du changement d'heure.

Nous pouvons aussi créer une variable de type date, à une date voulue :

```
var anniversaire = new Date(1998,04,12) ;  
console.log(anniversaire) ;
```

Voici l'affichage obtenu :



Le mois affiché en anglais est **May**, soit le mois de mai. Or, nous avons indiqué le mois ayant le numéro **4** ! Pourquoi ? Tout simplement parce que le mois de janvier, qui est le premier mois de l'année, possède l'index numéroté **0**, comme pour les tableaux. Donc le mois de décembre est numéroté **11**.

Si nous souhaitons avoir la date au mois d'avril, il faut donc indiquer la valeur **3**. Et nous pouvons aussi ajouter une heure, des minutes et des secondes.

```
var anniversaire = new Date(1998,03,12,15,30,23) ;  
console.log(anniversaire) ;
```

Voici l'affichage obtenu :



2. Des méthodes de l'objet Date()

À nouveau, nous n'allons aborder que quelques méthodes et propriétés de l'objet Date().

La méthode `getDay()` permet de connaître le numéro du jour dans la semaine d'une date donnée. Attention, le premier jour de la semaine est le **dimanche**, qui est numéroté **0**.

```
var anniversaire = new Date(1998,03,12,15,30,23) ;  
console.log(anniversaire.getDay()) ;
```

Voici l'affichage obtenu :



Dans cet exemple, le **12 avril 1998** était un **dimanche**. Le jour dimanche a bien pour numéro d'index **0**.

Voici un autre exemple d'utilisation de la méthode `getDay()` pour afficher le nom du jour.

Dans un fichier HTML, nous utilisons la fonction `switch()` :

```
<!DOCTYPE html>  
<html>  
  <head>  
    <meta charset="UTF-8">  
    <title>Mon script</title>  
  </head>  
  <body>  
    <script>  
      var anniversaire = new Date(1998,03,12,15,30,23) ;  
      var jourAnniversaire = anniversaire.getDay() ;  
      switch (jourAnniversaire) {  
        case 0:  
          alert("L'anniversaire était un dimanche");  
        case 1:  
          alert("L'anniversaire était un lundi");  
        case 2:  
          alert("L'anniversaire était un mardi");  
        case 3:  
          alert("L'anniversaire était un mercredi");  
        case 4:  
          alert("L'anniversaire était un jeudi");  
        case 5:  
          alert("L'anniversaire était un vendredi");  
        case 6:  

```

```

        alert("L\'anniversaire était un samedi");
    }
</script>
</body>
</html>

```

Voici l’affichage obtenu :



L’exemple à télécharger est dans le dossier **Chapitre-02-N/exemple-01.html**.

Nous pouvons aussi calculer le nombre de jours écoulés entre deux dates, avec la méthode `getTime()` et la méthode mathématique `ceil()`. Sachez que la méthode `getTime()` renvoie le nombre de millisecondes écoulées depuis le 1er janvier 1970.

Voici le code HTML et JavaScript de cet exemple :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <script>
      // Création de la fonction de calcul
      function differenceDates(date_1,date_2){
        var nbJours = date_2.getTime() - date_1.getTime();
        return Math.ceil(nbJours/(1000*60*60*24));
      }
      // Saisie des deux dates
      var date_1 = new Date(2018,01,01);
      var date_2 = new Date(2018,01,31);
      // Appel à la fonction
      alert(differenceDates(date_1,date_2) + " jours.");
    </script>
  </body>
</html>

```

Voici l’affichage obtenu :



La méthode `ceil()` de l’objet `Math()` permet de calculer le nombre de jours à partir d’une valeur en millisecondes.

L’exemple à télécharger est dans le dossier **Chapitre-02-J/exemple-02.html**.

3. Les propriétés et les méthodes de l'objet Date()

Voici la page du site des développeurs de Mozilla consacrée à l'objet Date() :

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

Le Document Object Model

1. Utiliser le DOM

Dans ce chapitre, nous allons aborder le DOM et voir comment accéder aux éléments constitutifs de la page, ceci afin d'utiliser JavaScript pour personnaliser l'affichage et le rendu de la page.

L'objectif de l'utilisation du JavaScript dans ce livre est la personnalisation de l'affichage et du rendu des pages web dans la fenêtre du navigateur. Pour atteindre cet objectif, nous devons connaître tous les constituants d'une page web, avec leurs propriétés, afin de pouvoir les exploiter.

Nous allons atteindre cet objectif avec le **Document Object Model**, plus connu sous son acronyme, le **DOM**.

Toutes les notions de base du code JavaScript que nous avons étudiées précédemment et tous les scripts s'exécutent au sein d'une page web. Cette page web, qu'elle soit simple ou très complexe, est toujours constituée d'éléments HTML et de règles CSS. C'est dans ce contexte qu'est exécuté le code JavaScript.

2. La notion de document

Le mot **Document** dans **Document Object Model (DOM)** représente le document, c'est-à-dire la page web. Il faut donc que nous sachions comment est constituée la page sur laquelle nous souhaitons intervenir.

Nous avons deux façons de voir une page web. Premièrement, par l'intermédiaire de son code source :

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <meta charset="UTF-8">
5      <title>Ma page</title>
6    </head>
7    <body>
8      <h1 id="titre-principal">Vulputate Sollicitudin Condimentum Bibendum</h1>
9      <h2 id="titre-secondaire">Bibendum Vestibulum Parturient</h2>
10     <p class="texte-courant">Integer posuere erat a ante venenatis dapibus posuere velit aliquet.
        Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Cras
        mattis consectetur purus sit amet fermentum. Curabitur blandit tempus porttitor. Donec sed
        odio dui. Cras justo odio, dapibus ac facilisis in, egestas eget quam. Vivamus sagittis lacus
        vel augue laoreet rutrum faucibus dolor auctor.</p>
11     <ul id="liste">
12       <li>Lorem</li>
13       <li>Ipsum</li>
14       <li>Sollicitudin</li>
15       <li>Vulputate</li>
16     </ul>
17     <p>Une image d'un zèbre :</p>
18     <p>
19     </p>
20   </body>
21 </html>
```

Et deuxièmement, par l'intermédiaire de son rendu dans les navigateurs :

Vulputate Sollicitudin Condimentum Bibendum

Bibendum Vestibulum Parturient

Integer posuere erat a ante venenatis dapibus posuere velit aliquet. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Cras mattis consectetur purus sit amet fermentum. Curabitur blandit tempus porttitor. Donec sed odio dui. Cras justo odio, dapibus ac facilisis in, egestas eget quam. Vivamus sagittis lacus vel augue laoreet rutrum faucibus dolor auctor.

- Lorem
- Ipsum
- Sollicitudin
- Vulputate

Une image d'un zèbre :



Mais pour JavaScript, une page web est juste une collection d'objets. Ce qui nous amène à la deuxième notion, celle d'objet.

3. La notion d'objet et d'arbre

Dans le chapitre précédent, nous avons largement abordé la notion d'objet en JavaScript. Rappelons donc juste qu'un objet est un élément possédant une collection de propriétés et de méthodes.

Donc, pour JavaScript, le document, la page web, est une collection d'objets. Chaque constituant de la page est un objet, donc tous les constituants d'une page sont des objets. Dans l'exemple ci-dessus, le titre de niveau 1, `<h1>`, est un objet, le paragraphe `<p>` est un objet, la liste non ordonnée `` est un objet. Mais il faut bien aussi considérer que les éléments `<body>` et `<html>` sont également des objets.

Ainsi, un objet peut être constitué d'autres objets. Des objets sont donc imbriqués dans d'autres objets. Par exemple, les éléments `` de la liste sont imbriqués dans l'élément ``. Le titre `<h1>` et le paragraphe `<p>` sont imbriqués dans l'élément `<body>`. Nous voyons donc se dessiner une imbrication d'objets sous la forme d'un **arbre hiérarchique**.

4. La notion de modèle

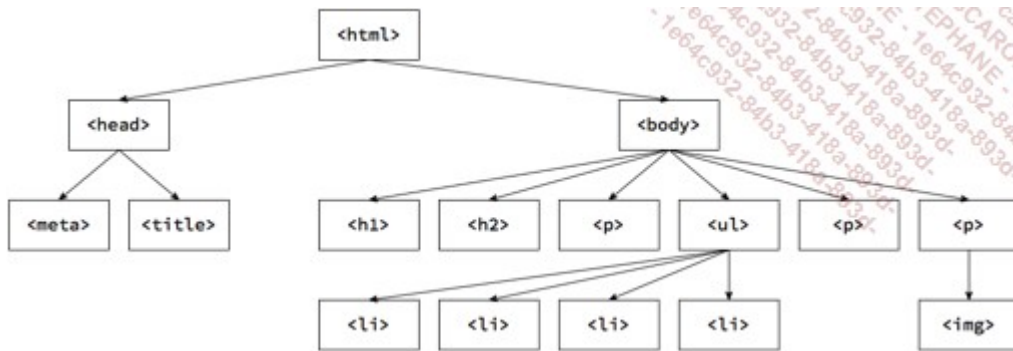
Pour décrire correctement cet arbre hiérarchique, il nous faut des termes qui soient standardisés dans une norme accessible et admise par tous. Cette notion de normalisation des termes constitue le **modèle**. C'est le troisième terme du **DOM, Model**.

Reprenons l'exemple de la page web précédente pour aborder des termes de la hiérarchie des objets.

- L'élément `<html>` est le parent de tous les autres éléments, qui y sont tous imbriqués. C'est l'élément **racine**.
- Chaque élément imbriqué est appelé un **nœud** (node en anglais).
- L'élément `` est le **parent** des éléments `` imbriqués.

- Chaque élément `` imbriqué est un **enfant** de l'élément `` **parent**.
- Chaque élément `` de la liste est un **frère** pour chaque autre élément ``.

Voici l'arbre hiérarchique des nœuds de cet exemple de page simple :



5. Le DOM

Nous pouvons donc dire que le **DOM** fait référence à trois notions distinctes, avec cette définition : le DOM est une normalisation qui permet de manipuler les objets constitutifs d'une page web.

- **Document** fait référence à la page web.
- **Object** fait référence aux objets constitutifs de la page web.
- **Model** fait référence aux termes normalisés qu'il faut utiliser.

Le DOM n'est donc pas une technologie, ni un langage, c'est une convention qui permet d'exploiter des objets avec la même normalisation. Le DOM est le lexique et la grammaire qui nous permet de manipuler le contenu des pages web.

Accéder aux éléments de la page web

1. La fenêtre du navigateur

L'exécution de tout le code JavaScript se fait dans le contexte de la fenêtre du navigateur. Cette fenêtre est représentée par l'objet `window`. L'objet `window` contient tout ce qui y est inclus, c'est-à-dire tous les éléments constitutifs, tous les nœuds accessibles avec le DOM, ainsi que l'environnement d'exécution du JavaScript.

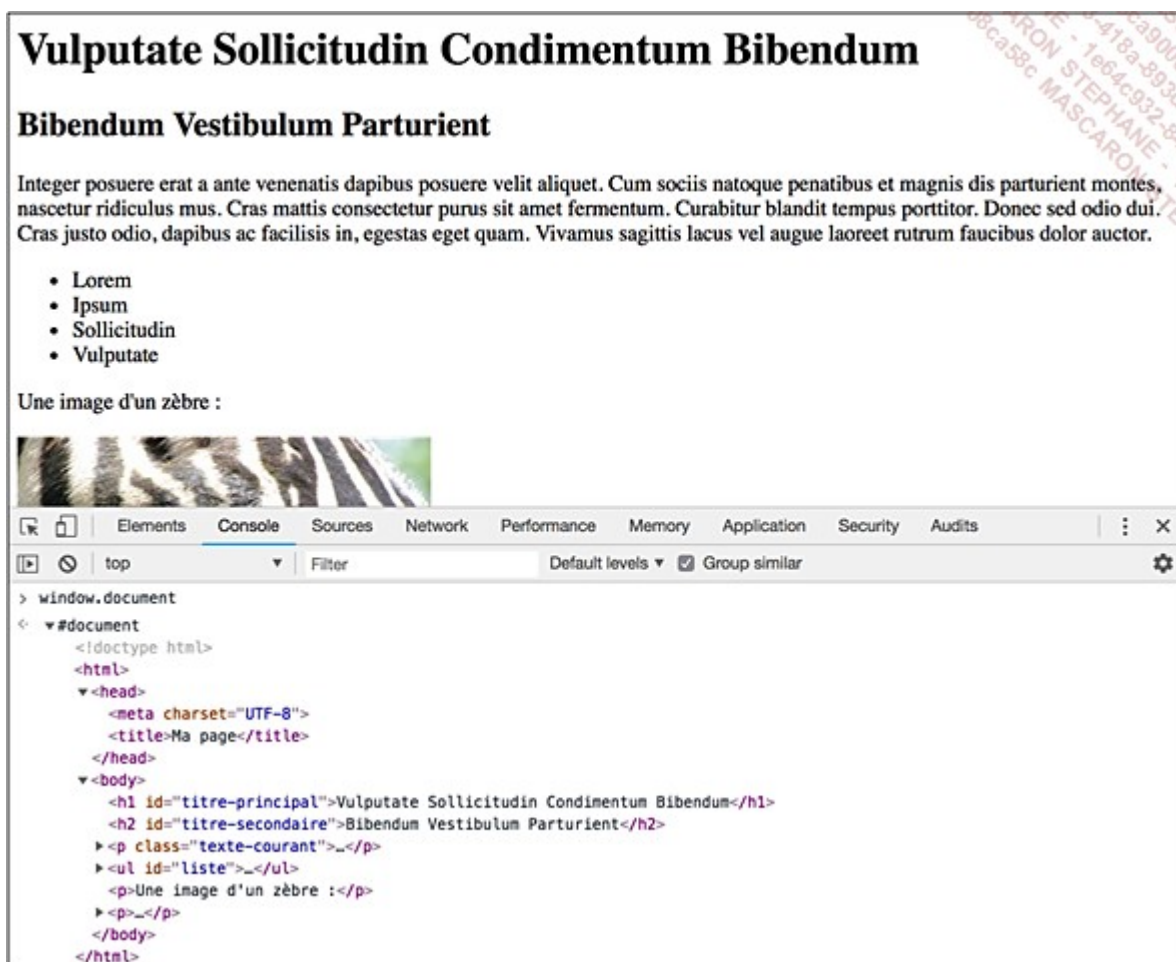
Souvenez-vous : lorsque nous avons abordé la notion de variable, nous avons évoqué les variables locales, qui n'étaient accessibles que dans les accolades de la fonction où elles sont définies, et les variables globales qui étaient accessibles partout dans le code JavaScript. Sachez donc que les variables globales sont définies comme des propriétés de l'objet `window`, elles sont en conséquence accessibles partout dans l'objet `window`, dans la fenêtre et dans l'environnement d'exécution du JavaScript. Notez qu'il n'est pas nécessaire de faire référence à l'objet `window` quand nous définissons une variable globale. C'est pour cela que vous ne verrez jamais le nom de la variable préfixée par `window`.

2. Le document

La page web est représentée par le document. Nous avons évoqué cette notion dans une section précédente. Sachez que le document est une propriété de l'objet window. Nous avons donc cette syntaxe pour accéder au document : `window.document`. Donc, tous les objets de `window.document` constituent l'arbre hiérarchique du DOM.

Si, dans la console de votre navigateur, vous saisissez `window.document`, la console va bien afficher tous les éléments, tous les objets constitutifs de la page. Tous ces objets sont les nœuds accessibles via le DOM.

Voici l'exemple avec le fichier HTML simple utilisé précédemment :



Pour déployer la structure de la page HTML dans la console, cliquez sur le triangle basculant de `#document`, puis faites de même avec les éléments imbriqués.

De même, si vous souhaitez accéder au nœud `<body>`, vous pouvez saisir `document.body`, sans qu'il soit nécessaire d'indiquer `window` puisqu'il est facultatif, car implicite.



Donc, chaque élément imbriqué dans la page est un objet qui est en fait une sous-propriété de `window.document`. Et à nouveau, chaque objet possède des propriétés et d'éventuels objets imbriqués.

Enfin, notez bien que `window.document` n'est pas qu'une simple représentation de la page web dans la fenêtre du navigateur. Non, `window.document` est une API très riche qui va nous permettre d'accéder à tous les éléments constitutifs de la page. Voici l'URL du site des développeurs de Mozilla sur les API JavaScript : <https://developer.mozilla.org/fr/docs/Web/API>.

3. Accéder aux nœuds de la page par les identifiants

L'objectif de l'utilisation du code JavaScript dans ce livre est de pouvoir modifier des éléments constitutifs de la page, de pouvoir en ajouter et en supprimer, de pouvoir modifier le contenu de certains éléments, d'appliquer des règles CSS, et bien d'autres fonctionnalités encore. Mais vous comprenez bien que pour mener à bien ces tâches, il faut que nous puissions accéder à ces éléments HTML et CSS. Pour ce faire, nous allons utiliser l'API de l'objet document.

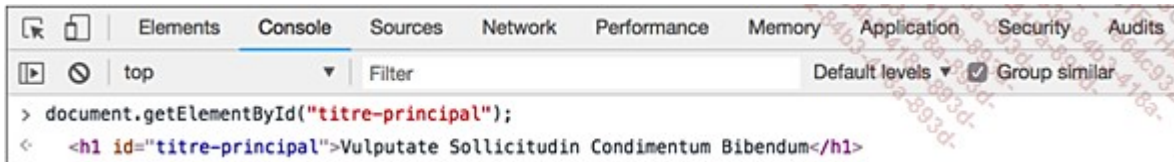
La méthode `getElementById()` de l'objet document permet d'accéder à un élément de la page si celui-ci possède un identifiant unique indiqué dans l'attribut `id`. L'identifiant à cibler est indiqué entre guillemets, dans les parenthèses de la méthode. Attention à bien respecter scrupuleusement le libellé de l'identifiant.

Dans notre exemple de page, le titre de niveau 1, `<h1>`, possède bien un attribut `id` :

```
<h1 id="titre-principal">Vulputate Sollicitudin Condimentum  
Bibendum</h1>
```

Donc dans la console, nous pouvons accéder à cet objet, à ce nœud du **DOM**, en indiquant : `document.getElementById("titre-principal");`. La console affiche cet élément ciblé par son identifiant : `"titre-principal"`.

Voici l’affichage obtenu :



Cette méthode est, bien sûr, la plus utilisée pour cibler un élément précis de la page web.

La méthode `getElementsByName()` permet de cibler un élément qui possède un attribut `name` renseigné. Par exemple :

```
<p name="premier">Vulputate Sollicitudin Condimentum Bibendum</p>
```

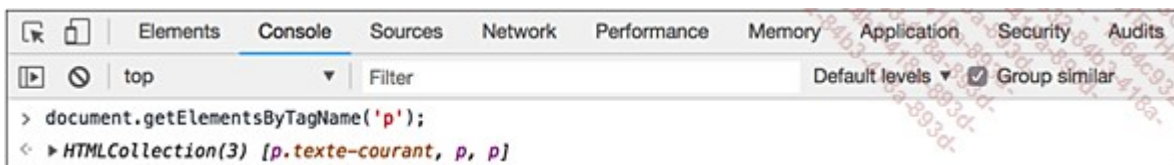
Cette méthode fonctionne de la même manière que la précédente, mais elle est moins utilisée.

4. Accéder aux nœuds de la page par les noms des éléments

Nous pouvons aussi utiliser la méthode `getElementsByTagName()` de l’objet `document`. Cette méthode permet de lister tous les éléments de la page, spécifiés par leur nom HTML. Notez bien la présence d’un S à la fin de `Elements`, car nous pouvons avoir plusieurs éléments du type spécifiés.

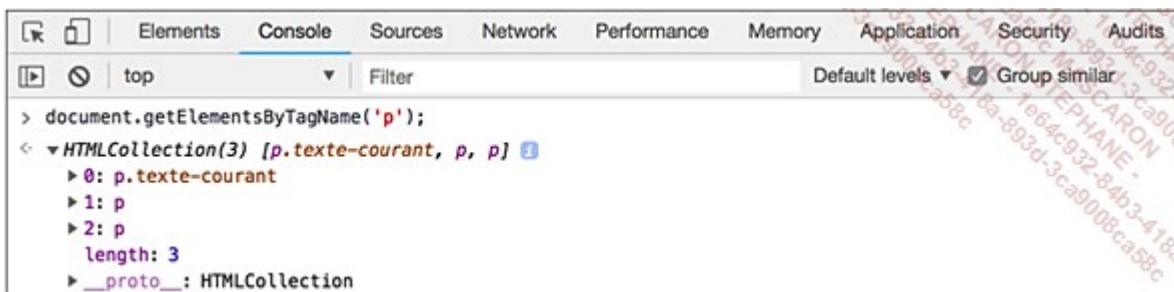
Dans cet exemple, nous allons afficher tous les éléments `<p>` de la page. Voici la syntaxe à utiliser : `document.getElementsByTagName('p');`.

Voici l’affichage obtenu :



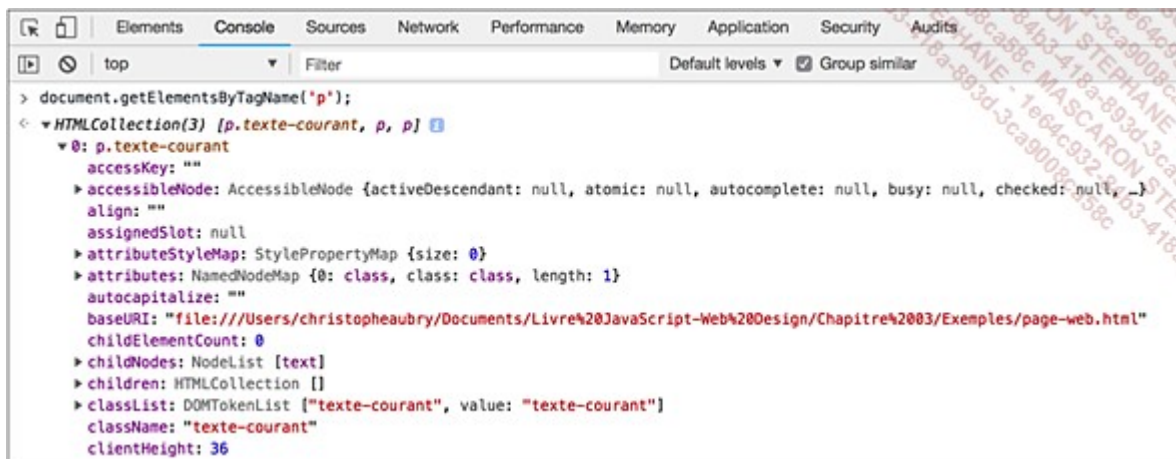
Le message de la console indique qu’il y a une collection de trois éléments HTML `<p>`. Cette collection est placée entre crochets, reprenant ainsi la syntaxe des tableaux.

Pour afficher plus de détails, cliquez sur le triangle basculant devant la collection :



Chaque élément de la liste est indexé. Le premier commence bien à l’index 0, comme dans les tableaux. La propriété `length` nous rappelle la longueur, le nombre d’éléments dans cette collection.

Si vous cliquez sur le triangle basculant d’un des éléments, vous affichez toutes ses propriétés :



À nouveau, il existe des triangles basculants pour chaque propriété qui peuvent afficher encore plus de détails.

Parmi ces propriétés, nous pouvons en noter quelques-unes :

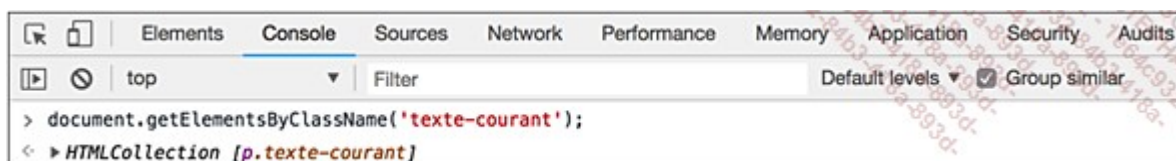
- `className: "texte-courant"` indique le nom de la classe.
- `id` indique la valeur de l'attribut d'identification de l'élément. Si cette valeur n'est pas renseignée, cela veut dire que l'élément n'a pas d'identifiant.
- `childNodes` indique la liste des nœuds enfants de l'élément, présentée sous la forme d'un tableau.
- `childElementCount: 0` indique que cet élément ne possède pas de nœud enfant, qu'il n'y a pas d'élément imbriqué.
- `nodeType: 1` indique que ce nœud est de type 1, c'est-à-dire un élément HTML.
- `nodeName: "P"` indique le nom HTML de l'élément. Ici, c'est un élément P, un paragraphe.
- `innerHTML` indique le contenu HTML et textuel de cet élément, mais aussi celui des autres éléments HTML éventuellement imbriqués.
- `innerText` n'indique que le contenu textuel de l'élément et celui des autres éléments HTML éventuellement imbriqués.

5. Accéder aux nœuds de la page par les noms de classe CSS

La méthode `getElementsByClassName` permet d'accéder à la collection de tous les éléments HTML qui utilisent une classe CSS spécifiée. Voici la syntaxe à utiliser :

```
document.getElementsByClassName('texte-courant');
```

Voici l'affichage obtenu :



Comme pour l'exemple précédent, vous pouvez afficher plus de détails avec les triangles basculants.


```

> document.getElementsByClassName('texte-courant');
< HTMLCollection [p.texte-courant]
  ▼ 0: p.texte-courant
    accessKey: ""
    ▶ accessibleNode: AccessibleNode {activeDescendant: null, atomic: null, autocomplete: null, busy: null, checked: null, ...}
    align: ""
    assignedSlot: null
    ▶ attributeStyleMap: StylePropertyMap {size: 0}
    ▶ attributes: NamedNodeMap {0: class, class: class, length: 1}
    autocapitalize: ""
    baseURI: "file:///Users/christopheaubry/Documents/Livre%20JavaScript-Web%20Design/Chapitre%2003/Exemples/page-web.html"
    childElementCount: 0
    ▶ childNodes: NodeList [text]
    ▶ children: HTMLCollection []
    ▶ classList: DOMTokenList ["texte-courant", value: "texte-courant"]
    className: "texte-courant"
  
```

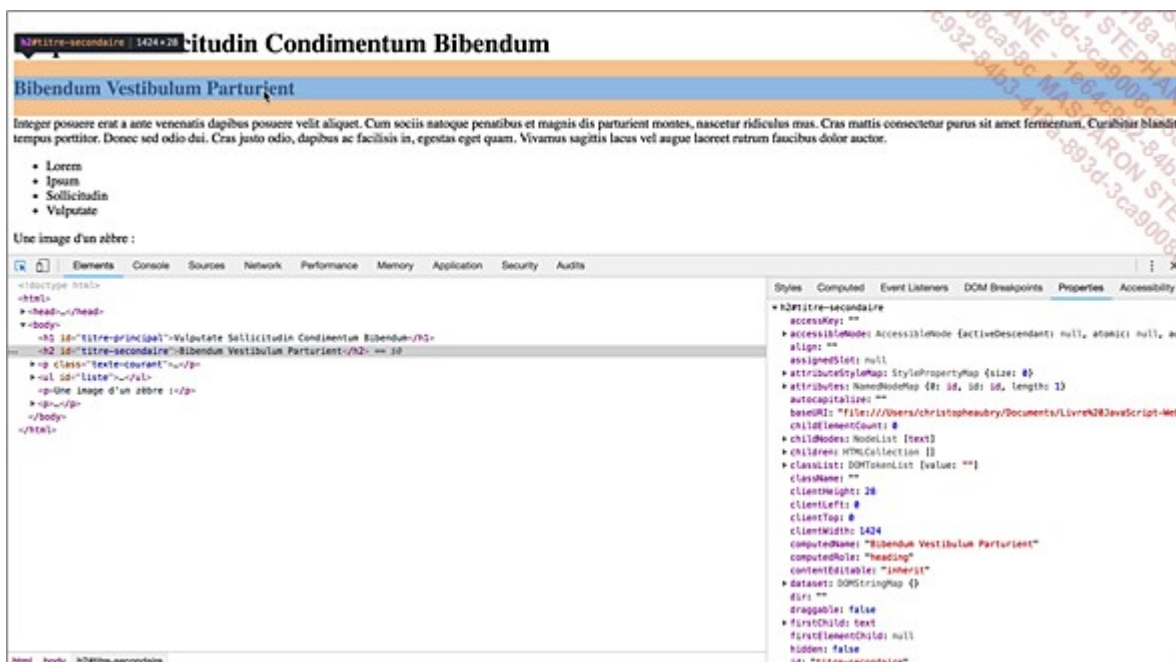
6. Afficher les propriétés d'un nœud

Nous venons de voir qu'il existe plusieurs méthodes pour accéder aux nœuds constitutifs de la page. Chaque nœud étant un objet, vous le savez maintenant, l'ensemble de ces nœuds possède de nombreuses propriétés que nous pouvons afficher et cibler en JavaScript.

Pour afficher les propriétés d'un objet avec les outils développeurs des navigateurs, il faut sélectionner l'objet voulu.

Dans cet exemple, avec Google Chrome, c'est l'objet `<h2 id="titre-secondaire"> Bibendum Vestibulum Parturient</h2>` qui est sélectionné. Dans l'onglet **Elements**, cet objet est bien sélectionné dans la liste des éléments constitutifs de la page.

Sur la droite du panneau, vous devez afficher l'onglet **Properties**. Puis, cliquez sur le triangle basculant devant `h2#titre-secondaire` :



Vous retrouvez la liste des très nombreuses propriétés de l'objet.

7. Accéder aux propriétés d'un nœud

Maintenant, nous souhaitons cibler une propriété spécifiée et afficher sa valeur dans la console. Pour ce faire, utilisons la syntaxe à point.

Dans cet exemple, nous souhaitons afficher le contenu textuel du dernier élément `` de la liste `<ul id="liste">`.

Voici le script inséré dans le fichier HTML :

```
<script>
  var maListe = document.getElementById('liste') ;
  var dernierLi = maListe.lastElementChild.innerText;
  console.log(dernierLi) ;
</script>
```

Analysons ce script simple :

- La variable `maListe` permet de récupérer le nœud dont l'identifiant est `liste`.
- Nous créons une nouvelle variable nommée `dernierLi`.
- Dans l'objet `maListe`, nous utilisons la propriété `lastElementChild` pour récupérer le dernier enfant de cet objet.
- Pour ce dernier enfant, nous utilisons la propriété `innerText` pour récupérer son contenu textuel.
- Nous affichons ce contenu dans la console.

Voici l'affichage obtenu : nous affichons bien le contenu textuel du dernier item de la liste.



Modifier des éléments dans la page web

1. Modifier un attribut HTML dans un élément

Dans ce premier exemple, pour un élément d'une page, nous allons modifier la valeur d'un de ses attributs HTML existants.

Voici l'élément HTML initial :

```
<h1 id="titre-principal" align="center">Vulputate Sollicitudin Condimentum  
Bibendum</h1>
```

Cet élément `<h1>` possède l'attribut `align`, avec la valeur `center`. Vous pouvez dire qu'il n'est pas très judicieux d'utiliser des attributs pour effectuer une mise en forme, qu'il vaut mieux utiliser des CSS. C'est parfaitement exact, mais l'objectif est ici de vous montrer l'utilisation d'une méthode du DOM.

Voici son affichage initial :



Notre objectif va être d'appliquer un alignement à droite. La première étape va être de cibler cet élément `<h1>` par son id. Ensuite, nous allons accéder à l'attribut `align`, pour en modifier la valeur.

Voici le script inséré dans le fichier HTML :

```
<script>  
    var titreH1 = document.getElementById('titre-principal') ;  
    titreH1.setAttribute('align','right');  
</script>
```

Étudions ce code très simple :

- Nous créons une variable nommée `titreH1`.
- Avec le DOM, nous accédons à l'élément HTML dont l'identifiant est `titre-principal`.
- Sur l'objet `titreH1`, nous utilisons la méthode `setAttribute()` des éléments du DOM.
- Le premier argument de cette méthode est le nom de l'attribut à modifier. C'est `align` dans cet exemple.
- Le deuxième argument est la valeur que nous souhaitons affecter à cet attribut, `right` dans cet exemple.

Voici l'affichage obtenu :



Voilà comment nous pouvons dynamiquement modifier la valeur d'un attribut d'un élément HTML identifié dans la page HTML.

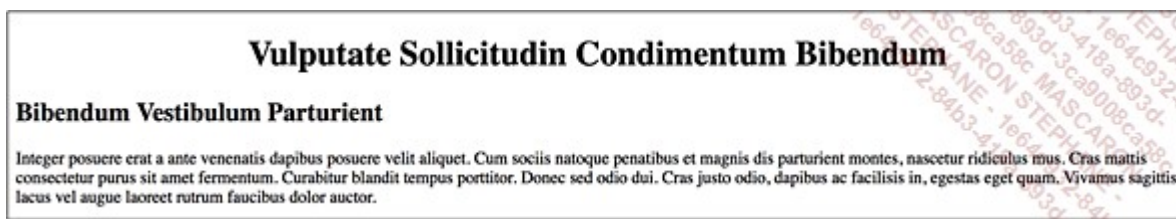
Voici la page consacrée à la méthode `setAttribute()` sur le site des développeurs Mozilla : <https://developer.mozilla.org/fr/docs/Web/API/Element/setAttribute>

2. Créer un attribut HTML dans un élément

La méthode `setAttribute()` permet aussi de créer un attribut, si celui-ci n'est pas présent.

Dans cet exemple, l'élément `<h2 id="titre-secondaire">Bibendum Vestibulum Parturient</h2>` n'a pas d'attribut `align`.

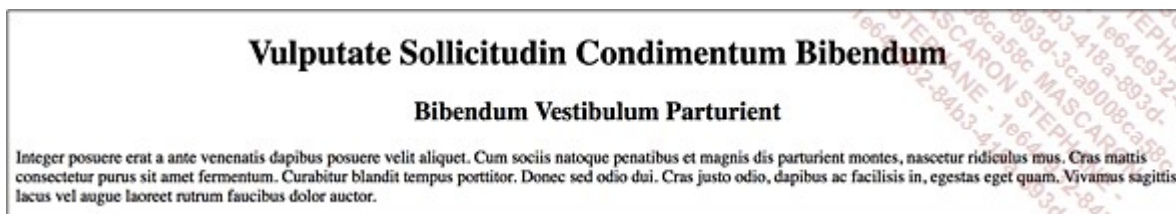
Voici son affichage initial :



Nous allons le créer et lui affecter une valeur, avec cette même méthode :

```
<script>
    var titreH2 = document.getElementById('titre-secondaire') ;
    titreH2.setAttribute('align','center');
</script>
```

Voici l'affichage obtenu :



3. Les propriétés `innerHTML` et `innerText`

Dans cet exemple, nous allons modifier le contenu d'un élément HTML de la page. Pour ce faire, nous allons utiliser la propriété `innerHTML` du DOM. Souvenez-vous que nous avons deux propriétés pour afficher le contenu des éléments HTML :

- La propriété `innerHTML` affiche tout le contenu de l'élément ciblé : le texte de cet élément, les balises et les contenus textuels des éventuels autres éléments HTML imbriqués dans cet élément.

- La propriété `innerText` n'affiche que le contenu textuel de l'élément et les autres contenus textuels des éventuels autres éléments HTML imbriqués dans cet élément.

Pour bien matérialiser cette différence, prenons l'exemple de cette liste :

```
<ul id="liste">
  <li>Lorem</li>
  <li>Ipsum</li>
  <li>Sollicitudin</li>
  <li>Vulputate</li>
</ul>
```

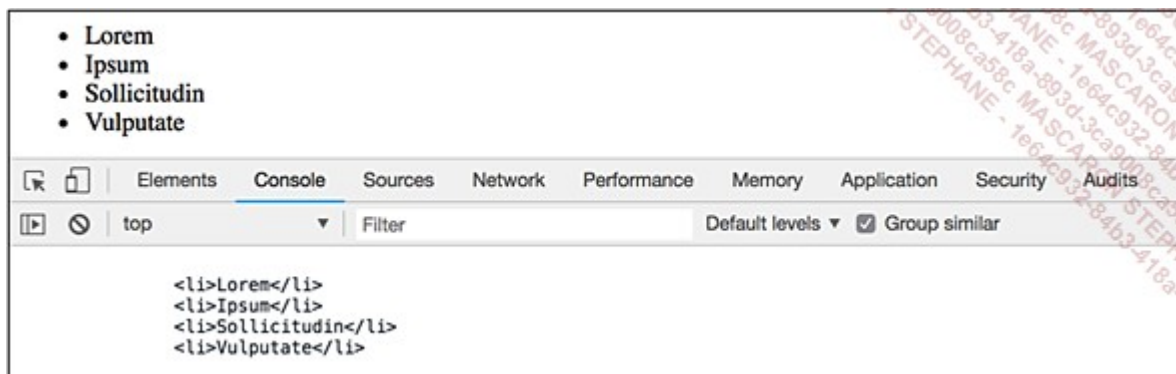
Voici le script utilisé avec la propriété `innerHTML` appliquée sur l'élément `` :

```
<script>
  var maListe = document.getElementById('liste') ;
  console.log(maListe.innerHTML) ;
</script>
```

Voici ce qu'affiche la console :

```
<li>Lorem</li>
<li>Ipsum</li>
<li>Sollicitudin</li>
<li>Vulputate</li>
```

Voici l'affichage obtenu dans le navigateur :



Nous avons bien l'affichage de toutes les balises HTML et de tous les textes.

Voici la page consacrée à cette propriété `innerHTML` sur le site des développeurs Mozilla :

<https://developer.mozilla.org/fr/docs/Web/API/Element/innerHTML>

Voici le script avec la propriété `innerText` appliquée sur l'élément `` :

```
<script>
  var maListe = document.getElementById('liste') ;
  console.log(maListe.innerText) ;
</script>
```

Voici ce qu'affiche la console :

```
Lorem
Ipsum
Sollicitudin
Vulputate
```

Voici l'affichage obtenu dans le navigateur :

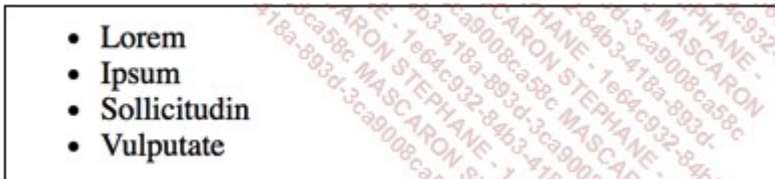


Nous avons bien uniquement les textes des éléments imbriqués dans l'élément ``

4. Modifier le contenu d'un élément de la page

Notre objectif dans cet exemple va être de modifier le contenu textuel du dernier item de la liste.

Voici l'affichage initial :

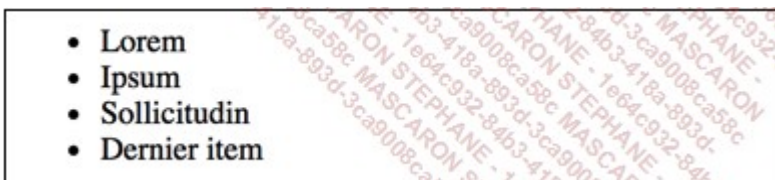


Voici le script inséré dans le fichier HTML :

```
<script>
  var maListe = document.getElementById('liste') ;
  var dernierLi = maListe.lastElementChild ;
  dernierLi.innerHTML = "Dernier item" ;
</script>
```

Ce script est très simple : après avoir ciblé le dernier élément enfant, `dernierLi`, avec `lastElementChild`, de la liste identifiée `maListe`, nous utilisons la propriété `innerHTML` pour modifier le contenu, avec le texte `Dernier item`.

Voici l'affichage obtenu :



5. Ajouter un contenu à un élément de la page

Si vous avez un élément HTML qui est vide de tout contenu, avec la méthode `innerHTML`, vous pouvez ajouter du contenu.

Voici un exemple très simple.

L'élément HTML vide de tout contenu :

```
<p id="vide"></p>
```

Le script :

```
<script>
    var monParaVide = document.getElementById('vide') ;
    monParaVide.innerHTML = "Nouveau contenu" ;
</script>
```

Voici l’affichage obtenu :

Nouveau contenu

Créer des éléments dans la page web

1. La première solution

Pour ce premier exemple, nous allons ajouter de nouveaux éléments HTML dans une page web, avec leur contenu textuel. Nous allons utiliser la méthode `createElement()`, qui permet de créer un nouvel élément HTML, et la méthode `appendChild()`, qui permet d’ajouter un nœud enfant à un nœud spécifié. Voici les pages consacrées à ces deux méthodes :

<https://developer.mozilla.org/fr/docs/Web/API/Document/createElement> et
<https://developer.mozilla.org/fr/docs/Web/API/Node/appendChild>

Voici la structure HTML initiale :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <div id="destinations">

    </div>
  </body>
</html>
```

Nous avons une boîte `<div>` qui possède un identifiant, `id="destinations"`, et qui est vide de tout contenu.

Voici le script inséré dans le fichier HTML, juste avant la balise `</body>` :

```
<script>
    // Création des deux nouveaux éléments
    var nouvelEnTeteH2 = document.createElement('h2') ;
    var nouvelleListe = document.createElement('ul') ;
    // Ajout du texte dans les deux nouveaux éléments
    nouvelEnTeteH2.innerHTML = "Nouvelles destinations" ;
    nouvelleListe.innerHTML =
"<li>Venise</li><li>Madrid</li><li>Londres</li>" ;
    // Localisation des ajouts
    document.getElementById('destinations').appendChild(nouvelEnTeteH2) ;
    document.getElementById('destinations').appendChild(nouvelleListe) ;
</script>
```

Analysons ce code.

La première étape permet de créer les nouveaux éléments HTML dans la page.

- Nous créons une variable nommée `nouvelEnTeteH2`.
- Cette variable permet la création, `createElement`, dans le document, `document`, d'un nouvel élément `h2` : `document.createElement('h2')`.
- Nous créons une deuxième variable nommée `nouvelleListe`.
- Cette variable permet la création dans le document d'un nouvel élément `ul` : `document.createElement('ul')`.

La deuxième étape consiste à ajouter du contenu aux deux variables créées précédemment.

- Pour chacune de ces deux variables, nous ajoutons du contenu HTML, avec la propriété `innerHTML` : `nouvelEnTeteH2.innerHTML` et `nouvelleListe.innerHTML`.
- Le premier contenu pour le `<h2>` est uniquement textuel : "Nouvelles destinations".
- Le deuxième contenu pour l'élément `` est constitué d'autres éléments imbriqués, des ``, et de leur contenu textuel : "`VeniseMadridLondres`".

La troisième étape consiste à indiquer où doivent être insérés les deux nouveaux éléments précédemment créés.

- Dans le document, `document`, nous ciblons par son identifiant `getElementById('destinations')` la boîte `<div>` où doivent être insérés les deux nouveaux éléments.
- Appliqués à cet élément identifié, nous ajoutons des nœuds enfants avec la méthode `appendChild()`.
- Ces deux nouveaux nœuds enfants ont pour contenu les variables indiquées entre les parenthèses de cette méthode.

Voici l'affichage obtenu :



Cette première solution utilise donc :

- La méthode `createElement()` pour créer de nouveaux éléments.
- La propriété `innerHTML` qui permet d'ajouter du texte, mais aussi d'autres éléments HTML imbriqués.
- La méthode `appendChild()` pour ajouter les nouveaux éléments enfants à un élément parent.

L'exemple à télécharger est dans le dossier **Chapitre-03-D/exemple-01.html**.

2. La deuxième solution

Nous allons reprendre la même structure HTML que précédemment. Nous allons ici utiliser une deuxième solution.

Voici le script utilisé :

```
<script>
  // Création des deux nouveaux éléments
  var nouvelEnTeteH2 = document.createElement('h2') ;
  var nouveauParagraphe = document.createElement('p') ;
  // Création des variables de texte
  var h2Texte = document.createTextNode("Nouvelles destinations") ;
  var pTexte = document.createTextNode("Venise, Madrid et Londres.") ;
  // Ajouter ces textes aux éléments créés
  nouvelEnTeteH2.appendChild(h2Texte) ;
  nouveauParagraphe.appendChild(pTexte) ;
  // Localisation des ajouts
  document.getElementById('destinations').appendChild(nouvelEnTeteH2) ;
  document.getElementById('destinations').appendChild(nouveauParagraphe) ;
</script>
```

Étudions ce code.

La première étape permet de créer les deux nouveaux éléments, comme précédemment.

La deuxième étape consiste à créer les contenus. Pour ce deuxième exemple, nous changeons sciemment de type de contenu. Dans le premier exemple, nous avons besoin de contenu texte et de balises HTML. Dans cet exemple, nous n'allons utiliser que du contenu textuel. C'est pour cela que nous utilisons la méthode `createTextNode()`. Cette méthode de l'objet `document` permet de créer un nouveau nœud de type texte. Voici l'URL de la page consacrée à cette méthode :

<https://developer.mozilla.org/fr/docs/Web/API/Document/createTextNode>. Le texte à insérer est placé entre guillemets et dans les parenthèses de la méthode. Cette solution est plus "propre" et plus technique, car elle utilise pleinement le DOM avec la création d'un nouveau nœud de type texte.

La troisième étape permet d'affecter avec `appendChild()`, aux nouveaux éléments `nouvelEnTeteH2` et `nouveauParagraphe`, les nœuds de texte `h2Texte` et `pTexte`.

La quatrième étape est similaire à la première solution.

Bien sûr, l'affichage obtenu est identique à la première solution.

Cette deuxième solution utilise donc :

- La méthode `createElement()` pour créer de nouveaux éléments.
- La méthode `createTextNode()` pour créer de nouveaux nœuds de texte.
- La méthode `appendChild()` pour ajouter les nœuds de texte aux nouveaux éléments et ensuite pour associer ces nouveaux éléments enfants à un élément parent.

Cette deuxième solution est un peu plus longue, mais est techniquement plus rigoureuse.

L'exemple à télécharger est dans le dossier **Chapitre-03-D/exemple-02.html**.

D'autres méthodes des nœuds

1. Supprimer un nœud enfant

La méthode `removeChild()` permet de supprimer un nœud enfant d'un nœud parent. Voici l'URL de la page consacrée à cette méthode :

<https://developer.mozilla.org/fr/docs/Web/API/Node/removeChild>

Voilà la structure simple de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body id="top">
    <h1>Vulputate Sollicitudin Condimentum Bibendum</h1>
    <p id="para">Donec sed odio dui</p>
  </body>
</html>
```

- L'élément `<body>` a un identifiant nommé `top`.
- Le paragraphe `<p>` a un identifiant nommé `para`.

Voici l'affichage initial :



L'objectif est de supprimer le paragraphe `<p id="para">`.

Ajoutons ce script très simple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body id="top">
    <h1>Vulputate Sollicitudin Condimentum Bibendum</h1>
    <p id="para">Donec sed odio dui</p>
  </body>
  <script>
    var parent = document.getElementById('top') ;
    var paragraphe = document.getElementById('para') ;
    var supp_para = parent.removeChild(paragraphe) ;
  </script>
</html>
```

Il nous faut dans un premier temps identifier le nœud parent du nœud enfant à supprimer. Dans cet exemple, l'élément parent du paragraphe `<p id="para">` est l'élément `<body id="top">`.

- Pour connaître cet élément parent, nous créons une variable nommée parent qui cible bien l'élément identifié avec la méthode getElementById('top').
- Ensuite, dans une autre variable, paragraphe, nous ciblons le nœud enfant avec getElementById('para').
- Enfin, dans une dernière variable, supp_para, nous supprimons le nœud enfant avec la méthode removeChild(), en utilisant la variable de stockage paragraphe.

Voici l'affichage obtenu :



L'exemple à télécharger est dans le dossier **Chapitre-03-E/exemple-01.html**.

2. Remplacer un nœud enfant

La méthode replaceChild() remplace un nœud enfant d'un nœud parent par un autre nœud.

Dans le fichier HTML initial, voici l'élément HTML qui sera remplacé : `<p id="para">Donec sed odio dui</p>` :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <h1>Vulputate Sollicitudin Condimentum Bibendum</h1>
    <p id="para">Donec sed odio dui</p>
  </body>
</html>
```

Voici l'affichage obtenu :



Voici le script utilisé :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <h1>Vulputate Sollicitudin Condimentum Bibendum</h1>
    <p id="para">Donec sed odio dui</p>
    <script>
```



```

// Création du nouvel élément pour le remplacement
var citation = document.createElement("blockquote") ;
// Création du nouveau contenu de texte
var citationTexte = document.createTextNode("Cras justo odio...") ;
// Ajout du contenu dans le nouvel élément
citation.appendChild(citationTexte) ;
// Cibler l'élément à remplacer
var paragraphe = document.getElementById("para") ;
// Créer la référence du parent
var paragrapheParent = paragraphe.parentNode ;
// Effectuer le remplacement
paragrapheParent.replaceChild(citation,paragraphe) ;
</script>
</body>
</html>

```

Analysons ce script :

- Nous créons un nouvel élément HTML de type `blockquote` dans la variable `citation`, avec la méthode `createElement()`.
- Nous créons dans la variable `citationTexte` un nœud de texte pour le contenu du nouvel élément, avec la méthode `createTextNode()`.
- Nous plaçons le texte, `citationTexte`, dans l'élément `citation` avec la méthode `appendChild()`.
- Nous plaçons dans la variable `paragraphe` l'élément à remplacer et identifié avec la méthode `getElementById()`.
- Dans la variable `paragrapheParent`, nous créons la référence de l'élément parent pour l'élément enfant `paragraphe`, avec la propriété `parentNode`.
- Enfin, avec l'élément parent `paragrapheParent`, nous remplaçons par le nouvel élément `citation` l'élément enfant `paragraphe`.

Donc, au final, le paragraphe `<p id="para">Donec sed odio dui</p>` sera remplacé par le nouvel élément `<blockquote>` et son contenu.

Voici l'affichage obtenu :



L'exemple à télécharger est dans le dossier **Chapitre-03-E/exemple-02.html**.

3. Dupliquer un nœud

La méthode `cloneNode(deep)` permet de dupliquer un nœud spécifié :

<https://developer.mozilla.org/fr/docs/Web/API/Node/cloneNode>. Une fois que le nœud est dupliqué, il ne fait pas encore partie du document, il faut l'ajouter au document avec la méthode `appendChild()`. Notez la présence de l'argument facultatif `deep`. S'il est à `true`, les nœuds enfants seront aussi dupliqués. S'il est à `false`, les nœuds enfants ne seront pas dupliqués.

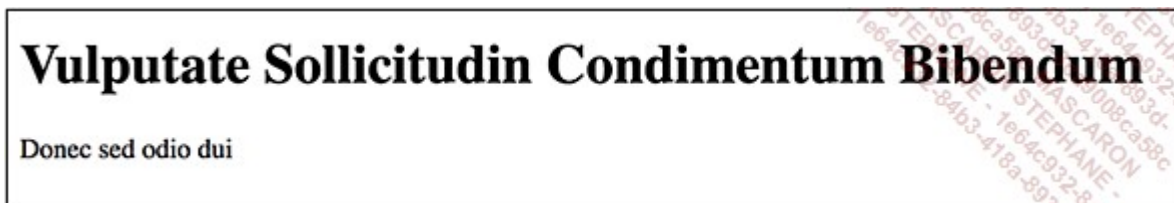


Attention, si vous dupliquez un nœud qui possède un id, ce dernier, comme tous les autres attributs, seront dupliqués. Donc, il convient de prendre des précautions afin d'éviter des conflits et des erreurs de code HTML, CSS et JavaScript.

Voici l'élément dans le fichier HTML, le nœud à dupliquer : `<p id="para">Donec sed odio dui.</p>` :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <h1>Vulputate Sollicitudin Condimentum Bibendum</h1>
    <p id="para">Donec sed odio dui</p>
  </body>
</html>
```

Voici l'affichage initial :



Voici le code JavaScript utilisé dans la page HTML :

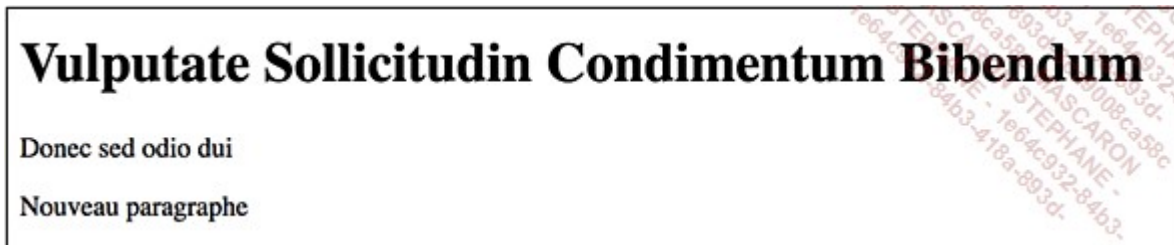
```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <h1>Vulputate Sollicitudin Condimentum Bibendum</h1>
    <p id="para">Donec sed odio dui</p>
    <script>
      var paraInit = document.getElementById("para") ;
      var paraDuplique = paraInit.cloneNode(false) ;
      paraDuplique.innerHTML="Nouveau paragraphe" ;
      paraDuplique.setAttribute('id','newPara');
      document.body.appendChild(paraDuplique) ;
    </script>
  </body>
</html>
```

Analysons ce code simple :

- Dans la première ligne, la variable `paraInit` identifie le nœud à dupliquer, `document.getElementById("para")`.
- Puis, le script crée une variable `paraDuplique` qui utilise la méthode `cloneNode()` pour dupliquer le nœud `paraInit` identifié précédemment.

- Ensuite, nous ajoutons du contenu dans le nœud dupliqué, avec la méthode `innerHTML`.
- Dans la quatrième ligne, nous prenons soin de modifier la valeur de l'attribut `id` du nœud dupliqué, afin d'éviter toute erreur. Nous utilisons la méthode `setAttribute()`. Le nouvel identifiant est `newPara` dans cet exemple.
- Enfin, nous ajoutons le nœud dupliqué à la fin du document, avec la méthode `appendChild()` appliquée à l'élément HTML `body`.

Voici l'affichage obtenu :



L'exemple à télécharger est dans le dossier **Chapitre-03-E/exemple-03.html**.

4. Insérer un nœud avant un nœud référence

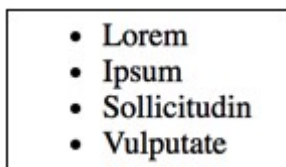
La méthode `insertBefore()` permet d'insérer un nœud enfant avant un nœud parent de référence : <https://developer.mozilla.org/fr/docs/Web/API/Node/insertBefore>

Dans cet exemple, nous voulons ajouter un nouvel item `` dans une liste ``.

Voici la structure HTML initiale :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Mon script</title>
  </head>
  <body>
    <ul id="liste">
      <li id="un">Lorem</li>
      <li id="deux">Ipsum</li>
      <li id="trois">Sollicitudin</li>
      <li id="quatre">Vulputate</li>
    </ul>
  </body>
</html>
```

Voici l'affichage initial :



Voici le code JavaScript utilisé :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
```

```

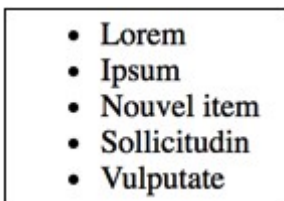
    <title>Mon script</title>
</head>
<body>
    <ul id="liste">
        <li id="un">Lorem</li>
        <li id="deux">Ipsum</li>
        <li id="trois">Sollicitudin</li>
        <li id="quatre">Vulputate</li>
    </ul>
    <script>
        var nouveauLi = document.createElement("li") ;
        nouveauLi.innerHTML = "Nouvel item" ;
        var liTrois = document.getElementById("trois") ;
        var parentUl = liTrois.parentNode ;
        parentUl.insertBefore(nouveauLi, liTrois);
    </script>
</body>
</html>

```

Étudions ce code :

- Dans la variable `nouveauLi`, nous créons un nouvel élément HTML ``.
- Dans ce nouvel élément, nous plaçons le contenu textuel "Nouvel item".
- Dans la variable `liTrois`, nous ciblons l'élément dont l'identifiant est `id="trois"`. Cela correspond au troisième item de la liste. C'est avant cet item que doit être inséré le nouvel item créé précédemment.
- Dans la variable `parentUl`, nous référençons l'élément parent du nouvel élément enfant.
- Enfin, pour le nœud parent, avec la méthode `insertBefore()`, nous insérons le nouveau nœud enfant, `nouveauLi`, avant l'élément `liTrois`.

Voici l'affichage obtenu :



L'exemple à télécharger est dans le dossier **Chapitre-03-E/exemple-04.html**.

5. Tester la présence de nœuds enfants

Avant d'exécuter une action, il peut être opportun de savoir si un nœud possède des nœuds enfants. C'est l'objectif de la méthode `hasChildNodes()` :

<https://developer.mozilla.org/fr/docs/Web/API/Node/hasChildNodes>

Reprenons la structure précédente :

```

<ul id="liste">
    <li id="un">Lorem</li>
    <li id="deux">Ipsum</li>
    <li id="trois">Sollicitudin</li>
    <li id="quatre">Vulputate</li>
</ul>

```

Nous allons tester si l'élément possède des nœuds enfants.

Voici le script :

```
<script>
  var maListe = document.getElementById("liste") ;
  var enfants = maListe.hasChildNodes() ;
  console.log(enfants) ;
</script>
```

Analysons ce script simple :

- Dans la variable maListe, nous référençons le nœud parent cible.
- Dans la variable enfants, nous stockons le résultat du test, hasChildNodes(), qui détermine si le nœud parent, enfants, possède des enfants.
- Nous affichons le résultat, true ou false, dans la console.

Voici l'affichage obtenu :



L'élément parent possède bien des enfants .

L'exemple à télécharger est dans le dossier **Chapitre-03-E/exemple-05.html**.

Comprendre les événements

Tout le code JavaScript que nous avons vu dans les chapitres précédents ne s'exécute qu'une seule fois, au chargement de la page. C'est donc une utilisation extrêmement limitée, il n'y a aucune interaction avec l'utilisateur de la page. Un des objectifs du JavaScript est de dynamiser la page, de proposer des interactions avec les utilisateurs.

Or, comment interagir avec ces utilisateurs ? Eh bien, par l'intermédiaire d'événements. Les événements se produisent à tout moment dans une page, par exemple lorsque le pointeur de la souris se déplace sur la page, quand il passe au-dessus d'un élément et quand il en sort, quand l'utilisateur clique sur un bouton. Mais vous avez aussi des événements qui sont déclenchés avec le clavier, comme appuyer sur une touche ou relâcher cette touche du clavier. Donc, il y a en permanence des événements qui se produisent dans une page web.

Aussi, pour nous, la seule question qu'il faut se poser est : à quels événements nous devons réagir et de quelle manière ? Pour réagir à un événement, il faudra être à son écoute, il faudra gérer un écouteur spécifique à cet événement.

Utiliser un bouton déclencheur d'action

1. Réagir au clic sur un bouton

Dans cet exemple très simple, nous allons réagir au clic de l'utilisateur sur un bouton de la page.

Voici la structure de la page et celle du bouton :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <p>Integer posuere erat a ante venenatis...</p>
    <button id="monBouton">Cliquez-moi</button>
  </body>
</html>
```

Le bouton possède l'identifiant id="monBouton".

Voilà le code JavaScript utilisé dans cette page :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <p>Integer posuere erat a ante venenatis...</p>
    <button id="monBouton">Cliquez-moi</button>
    <script>
      var leBouton = document.getElementById("monBouton") ;
      leBouton.onclick = function(){
        alert('Vous avez cliqué sur le bouton.') ;
      }
    </script>
  </body>
</html>
```

```

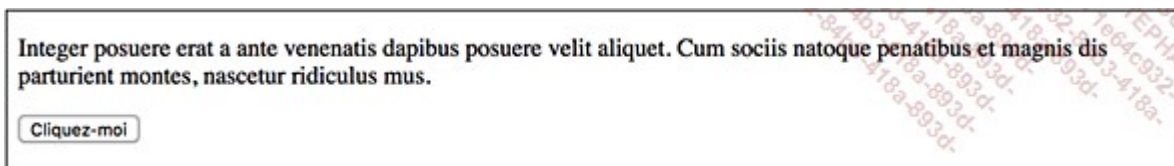
    } ;
  </script>
</body>
</html>

```

Détaillons ce script simple :

- Dans la variable leBouton, nous identifions le bouton.
- Ensuite, nous associons à ce bouton l'événement onclick. Cela permet de savoir si l'utilisateur clique sur ce bouton.
- Si l'utilisateur clique sur le bouton, nous exécutons une fonction anonyme : `function(){...}`. Cette fonction est dite anonyme, car elle ne possède pas de nom. Ce n'est pas nécessaire, car cette fonction n'est appelée qu'une seule fois, qu'à un seul moment, lorsque l'utilisateur clique sur le bouton. Cette fonction n'a pas besoin d'être appelée ailleurs.
- Cette fonction affiche un simple message dans un `alert()`.

Voici la page initiale :



Voici l'affichage obtenu lorsque l'utilisateur clique sur le bouton :



Nous avons bien l'affichage du message d'alerte lorsque l'utilisateur clique sur le bouton.

L'exemple à télécharger est dans le dossier **Chapitre-04-A/exemple-01.html**.

2. Utiliser un écouteur d'événement

Dans ce deuxième exemple, nous allons utiliser la fonction JavaScript écouteur d'événements : `addEventListener()`,

<https://developer.mozilla.org/fr/docs/Web/API/EventTarget/addEventListener>

Nous allons utiliser la même structure de page que dans l'exemple précédent :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <p>Integer posuere erat a ante venenatis...</p>
    <button id="monBouton">Cliquez-moi</button>
  </body>
</html>

```


Voilà le code JavaScript utilisé dans cet exemple :

```
<script>
  var leBouton = document.getElementById("monBouton") ;
  leBouton.addEventListener("click",monAlerte,false) ;
  function monAlerte(){
    alert('Vous avez cliqué sur le bouton.') ;
  } ;
</script>
```

Analysons ce code :

- Nous retrouvons la même variable pour identifier le bouton.
- Sur ce bouton, nous utilisons la méthode `addEventListener()`.
- Le premier argument est l'événement que nous souhaitons écouter. Cet argument est une chaîne de caractères, il est donc placé entre guillemets. C'est l'événement "click", c'est le fait que l'utilisateur clique sur un élément identifié. Voici l'URL des événements que nous pouvons écouter : <https://developer.mozilla.org/fr/docs/Web/Events>
- Le deuxième argument est le nom de la fonction qui doit être appelée lorsque cet événement est déclenché : `monAlerte`.
- Le troisième argument concerne des options, nous n'en avons pas besoin, donc nous indiquons `false`.
- Ensuite, nous avons la définition de la fonction `monAlerte()` qui doit être exécutée.

Les affichages sont strictement identiques aux précédents.

Cette deuxième solution est nettement plus propre et plus technique. Elle distingue bien tous les paramètres liés à l'écoute d'un événement.

L'exemple à télécharger est dans le dossier **Chapitre-04-A/exemple-02.html**.

Utiliser d'autres événements

1. Les événements à écouter

Nous avons de nombreux événements à écouter à notre disposition :

<https://developer.mozilla.org/fr/docs/Web/Events>

Voici ceux liés au pointeur de la souris : <https://developer.mozilla.org/fr/docs/Web/API/MouseEvent>

Nous n'allons bien sûr pas les passer tous en revue, ils sont simples à mettre en œuvre.

2. Le pointeur "au-dessus"

Le premier événement que nous allons aborder, c'est le simple fait de passer le pointeur de la souris au-dessus d'un élément identifié. C'est l'événement `mouseover`.

Voici l'élément ciblé, un simple paragraphe ayant l'identifiant `para` :

```
<!DOCTYPE html>
<html>
```

```

<head>
  <meta charset="UTF-8">
  <title>Ma page</title>
</head>
<body>
  <p id="para">Integer posuere erat a ante venenatis...</p>
</body>
</html>

```

Et voici le code JavaScript utilisé :

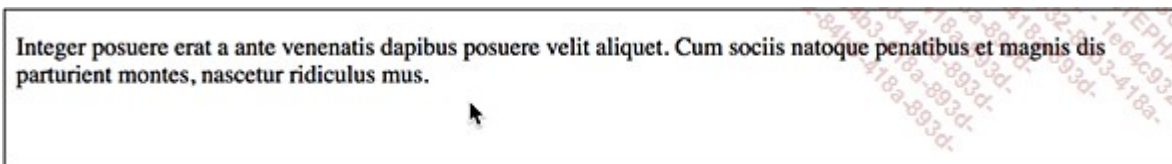
```

<script>
  var monElement = document.getElementById("para") ;
  monElement.addEventListener("mouseover", monAlerte, false) ;
  function monAlerte(){
    alert('Événement détecté.') ;
  } ;
</script>

```

Nous avons le même code que précédemment. Le seul élément qui change est l'événement écouté, `mouseover` dans cet exemple.

Voici l'affichage initial, avec la souris qui n'est pas au-dessus du paragraphe :



Voici l'affichage obtenu lorsque le pointeur de la souris passe au-dessus du paragraphe `<p id="para">` :



L'événement `mouseout` est déclenché lorsque le pointeur sort de la zone de l'élément identifié.

L'exemple à télécharger est dans le dossier **Chapitre-04-C/exemple-01.html**.

3. Événement lié aux formulaires

Les formulaires ont aussi leurs événements. Nous allons dans cet exemple utiliser l'événement `onblur`, qui est déclenché lorsque le point d'insertion quitte un champ.

Voici la structure du formulaire que nous allons utiliser :

```

<form id="inscription" name="formulaire" method="post" action="#">
  <p>
    <label for="nom">Nom : </label>
    <input id="nom" name="nom" type="text">
  </p>
  <p>
    <label for="prenom">Prénom : </label>
    <input id="prenom" name="prenom" type="text">
  </p>

```

```

<p>
  <label for="motDePasse">Mot de passe : </label>
  <input id="motDePasse" name="motDePasse" type="password">
</p>
<p>
  <input type="submit" name="envoyer" id="envoyer" value="Envoyer">
</p>
</form>

```

C'est le champ du mot de passe qui nous intéresse, il est identifié par `id="motDePasse"`. L'objectif est d'afficher un message d'alerte si ce champ contient moins de 8 caractères.

Voici le code utilisé :

```

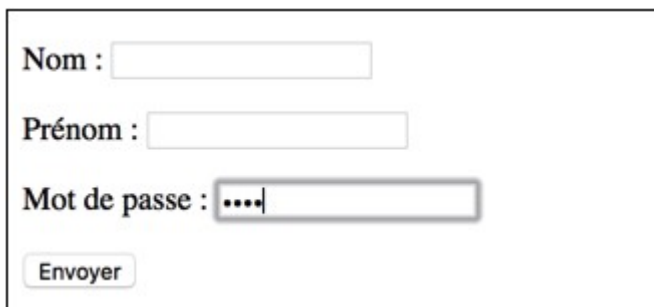
<script>
  var champMDP = document.getElementById("motDePasse") ;
  champMDP.onblur = function () {
    var nbCar = champMDP.value.length ;
    if ( nbCar < 8 ) {
      alert ("Le mot de passe doit contenir au moins 8 caractères.")
    }
  };
</script>

```

Analysons ce code :

- Nous créons une variable nommée `champMDP` pour identifier le champ du mot de passe.
- Lorsque le point d'insertion quitte, `onblur`, ce champ `champMDP`, nous exécutons une fonction anonyme.
- Dans cette fonction, nous créons une variable `nbCar` qui compte le nombre de caractères saisis dans le champ du mot de passe : `champMDP.value.length`.
- Ensuite, nous testons si ce nombre est inférieur à 8 : `if (nbCar < 8)`.
- Si c'est le cas, nous affichons une alerte.

Voici le champ du mot de passe avec moins de 8 caractères saisis :



Nom :

Prénom :

Mot de passe :

Si nous passons au champ suivant, le message d'alerte est affiché :



Nom :

Prénom :

Mot de passe :

Cette page indique

Le mot de passe doit contenir au moins 8 caractères.

Les formulaires permettent d'utiliser d'autres événements :

- onfocus : lorsque le pointeur rentre dans un champ.
- focus : lorsqu'un champ possède le focus, c'est-à-dire qu'il est actif.
- focusin : lorsqu'un champ va recevoir le focus.
- focusout : lorsqu'un champ va perdre le focus.

L'exemple à télécharger est dans le dossier **Chapitre-04-C/exemple-02.html**.

4. L'événement de chargement

Lorsque la fenêtre du navigateur est prête à être utilisée, un événement est déclenché. Il s'agit de onload. Cet événement est donc déclenché avant le chargement de la page dans la fenêtre du navigateur.

Reprenons la structure précédente, avec le formulaire. Dans l'élément <script> de cette page, nous utilisons l'objet window, auquel nous appliquons l'événement onload. Et lorsque cet événement est déclenché, nous affichons un message d'alerte.

Voici ce simple code :

```
<script>
    window.onload = function(){
        alert("Page chargée !")    ;
    };
</script>
```

Voici l'affichage obtenu lorsque la fenêtre du navigateur est prête à être utilisée, donc avant que la page et ses éléments constitutifs soient chargés et affichés :



Lorsqu'ensuite l'utilisateur clique sur le bouton de validation du message d'alerte, la page se charge et s'affiche :

L'exemple à télécharger est dans le dossier **Chapitre-04-C/exemple-03.html**.

Introduction CSS et JavaScript

Rappelons que l'un des objectifs du code JavaScript est de dynamiser les pages web, d'ajouter de l'interaction avec les visiteurs. Eh bien, nous allons pouvoir utiliser JavaScript pour interagir sur les CSS, suite à des événements utilisateur, par exemple.

Accéder aux propriétés CSS

1. Les CSS en ligne

Nous avons vu précédemment comment accéder à de nombreuses propriétés des objets via JavaScript. Les propriétés CSS ne dérogent pas à cette règle.

Voici un élément HTML simple :

```
<h1 id="titre1">Vulputate Sollicitudin Condimentum Bibendum</h1>
```

Pour accéder aux propriétés CSS de cet objet, nous allons utiliser la propriété style de l'objet HTMLelement.

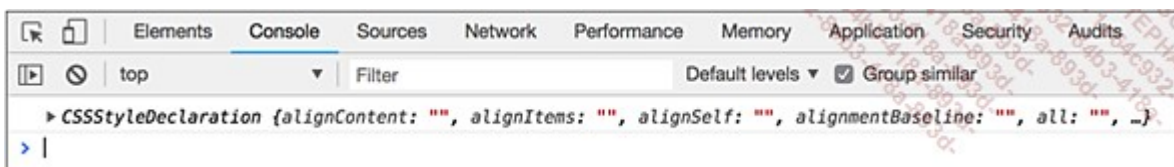


Attention, cette propriété ne permet d'accéder qu'aux styles CSS en ligne des éléments ciblés, c'est-à-dire aux propriétés CSS déclarées dans l'attribut style="..." de l'élément. Nous n'accédons pas aux styles CSS déclarés dans l'élément <style> ou ceux déclarés dans un fichier .css lié à la page.

Pour accéder aux propriétés CSS de cet élément et pour les afficher dans la console, voici le script très simple utilisé :

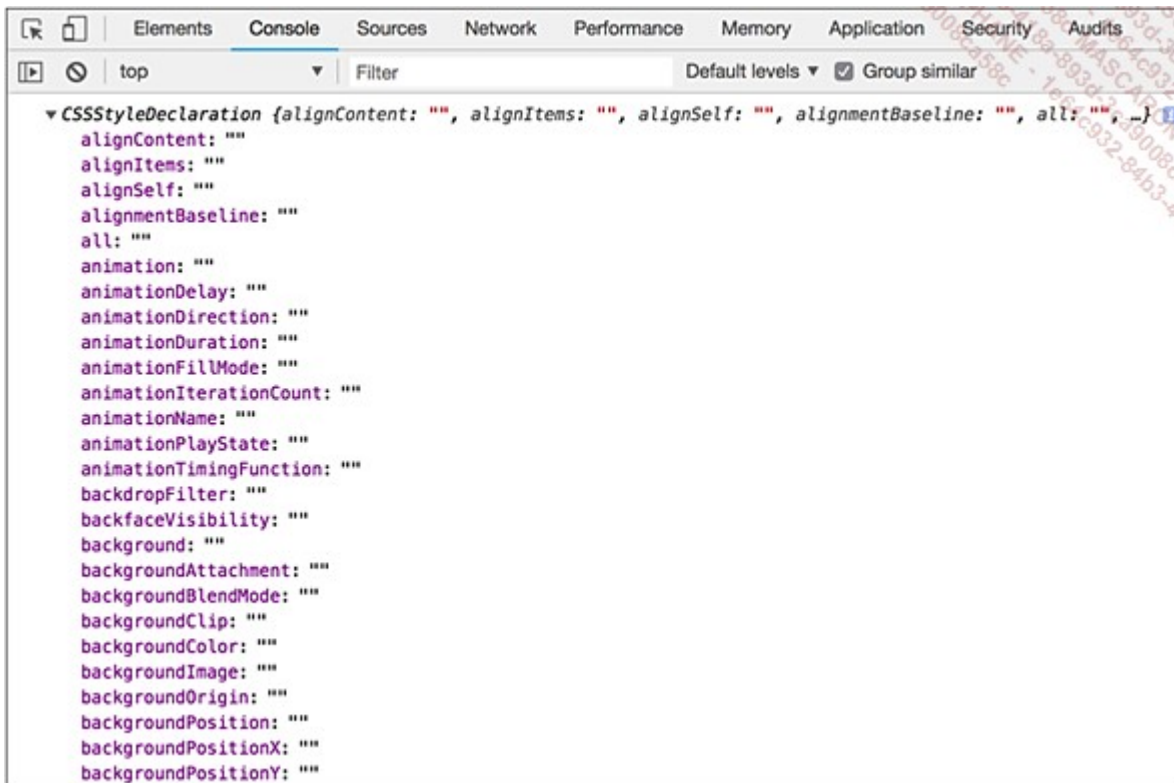
```
<script>
    var monTitre1 = document.getElementById("titre1") ;
    console.log(monTitre1.style) ;
</script>
```

Voici l'affichage obtenu :



La console affiche la déclaration CSSStyleDeclaration {...} qui contient toutes les propriétés CSS de l'élément HTML ciblé.

Pour lister ces propriétés, cliquez sur le triangle basculant devant CSSStyleDeclaration {...}.



Bien sûr, toutes ces propriétés sont vides de valeur, pour le moment.

Maintenant, nous allons ajouter des styles CSS en ligne dans notre élément :

```
<h1 id="titre1" style="color: #FFCC22; background-color: #dedede;">Vulputate Sollicitudin Condimentum Bibendum</h1>
```

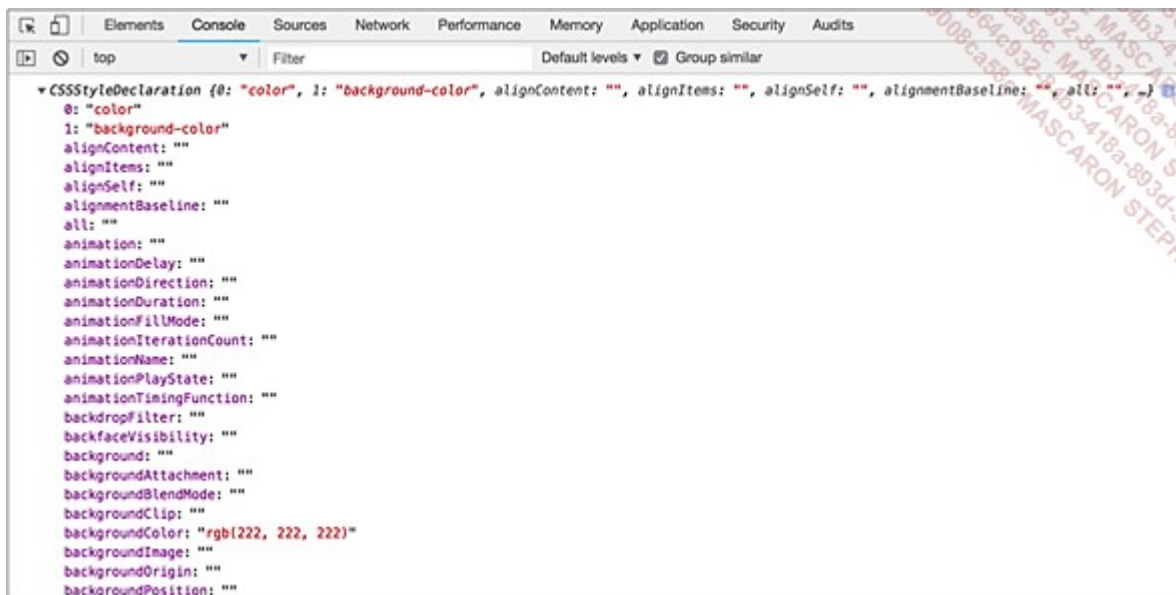
Et à nouveau, nous affichons dans la console les propriétés CSS de ce titre <h1>, avec le même script que précédemment. Voici l’affichage obtenu :



Dans la déclaration `CSSStyleDeclaration {}`, nous avons maintenant deux propriétés qui sont indexées à 0 et 1 et elles sont affichées en rouge. C’est normal, car ces deux propriétés sont utilisées dans l’élément source.

2. La syntaxe des propriétés

Vous remarquez certainement des particularités au niveau de la syntaxe des propriétés CSS. Si nous ouvrons la déclaration des CSS en cliquant sur le triangle basculant devant la déclaration `CSSStyleDeclaration {}`, nous affichons toutes les propriétés CSS :



La propriété CSS standard pour appliquer une couleur d'arrière-plan à un élément s'appelle background-color. Or, nous avons vu dans un chapitre précédent que JavaScript n'accepte pas le tiret - dans sa syntaxe. C'est pour cette raison qu'en JavaScript cette propriété est renommée backgroundColor. Il en est de même avec toutes les propriétés CSS utilisant un tiret ; par exemple, font-weight devient fontWeight. En JavaScript, le tiret est supprimé et le deuxième mot prend une majuscule.

Appliquer des styles CSS

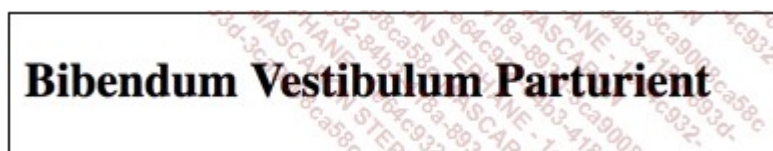
1. Appliquer des styles en ligne

Dans cet exemple, nous allons appliquer des styles CSS à un élément ciblé. Les styles appliqués seront placés dans la balise d'ouverture de l'élément, dans l'attribut style.

Voici la page HTML, avec l'élément <h2> ciblé qui ne possède pas de style :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <h2 id="titre2">Bibendum Vestibulum Parturient</h2>
  </body>
</html>
```

Voici l'affichage initial :



Voici le code JavaScript utilisé :

```
<!DOCTYPE html>
```



```

<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <h2 id="titre2">Bibendum Vestibulum Parturient</h2>
    <script>
      var monTitre2 = document.getElementById("titre2") ;
      monTitre2.style.color = "#FF0000" ;
      monTitre2.style.backgroundColor = "silver" ;
    </script>
  </body>
</html>

```

Analysons ce code très simple :

- Dans la variable monTitre2, nous référençons l'élément <h2> ciblé.
- Pour l'élément monTitre2, avec le sous-objet style, nous appliquons à la propriété CSS color la valeur #FF0000, soit une couleur rouge au texte.
- Nous ajoutons une couleur argentée, silver, en arrière-plan.



Notez bien la syntaxe JavaScript de la propriété d'arrière-plan backgroundColor.

Voici l'affichage obtenu :

Bibendum Vestibulum Parturient

Affichez le code source et visualisez l'application de ces propriétés CSS dans l'attribut style :

```

<h2 id="titre2" style="color: rgb(255, 0, 0);
background-color: silver;">Bibendum Vestibulum Parturient</h2>

```

Vous remarquez que la syntaxe hexadécimale de la couleur du texte a été dynamiquement transformée en syntaxe rgb().

L'exemple à télécharger est dans le dossier **Chapitre-05-C/exemple-01.html**.

2. Appliquer une classe CSS

Il se peut que vous ayez déjà une règle CSS prête à être utilisée, mais pas encore appliquée à l'élément HTML voulu. Nous allons pouvoir attribuer à cet élément HTML la classe CSS voulue, et les propriétés CSS seront ainsi appliquées.

Voici la structure HTML de la page de cet exemple :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <link href="styles.css" rel="stylesheet">
    <style>
      .texte-important {

```



```
<p id="conclusion" class="texte-important">Integer posuere erat a ante venenatis...</p>
```

Nous avons bien l'attribut class avec la valeur "texte-important".



Notez que cette solution fonctionne aussi parfaitement bien avec des classes CSS qui sont enregistrées dans un fichier .css lié au fichier .html.

L'exemple à télécharger est dans le dossier **Chapitre-05-C/exemple-02.html**.

3. Modifier des styles CSS

Il est possible qu'un élément HTML possède déjà une classe CSS qui lui applique des propriétés CSS, mais vous souhaitez modifier celles-ci.

Voici la structure HTML et CSS initiale :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <link href="styles.css" rel="stylesheet">
    <style>
      .texte-important {
        background-color: #dedede ;
        color: #ff0000;
        border-left: 5px solid #000;
        padding-left: 10px;
      }
    </style>
  </head>
  <body>
    <p id="conclusion" class="texte-important">Integer
posuere erat a ante venenatis dapibus...</p>
  </body>
</html>
```

Le paragraphe <p id="conclusion"> utilise la classe class="texte-important" qui est déclarée dans la page. Cette classe applique une couleur d'arrière-plan grise, une couleur de texte rouge, une bordure à gauche et un remplissage à gauche.

Voici l'affichage initial :

Integer posuere erat a ante venenatis dapibus posuere velit aliquet. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Cras mattis consectetur purus sit amet fermentum. Curabitur blandit tempus porttitor. Donec sed odio dui. Cras justo odio, dapibus ac facilisis in, egestas eget quam. Vivamus sagittis lacus vel augue laoreet rutrum faucibus dolor auctor.

Voici le script qui va permettre de modifier les styles CSS existants et qui va ajouter un nouveau style :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
```

```

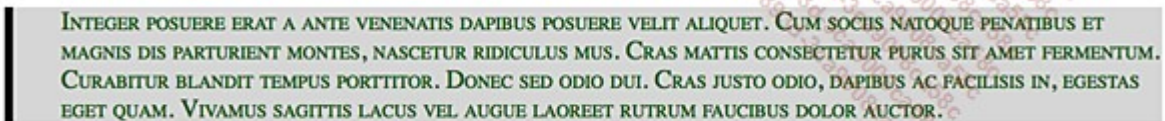
<title>Ma page</title>
<link href="styles.css" rel="stylesheet">
<style>
    .texte-important {
        background-color: #dedede ;
        color: #ff0000;
        border-left: 5px solid #000;
        padding-left: 10px;
    }
</style>
</head>
<body>
    <p id="conclusion" class="texte-important">Integer
posuere erat a ante venenatis dapibus...</p>
    <script>
        var maConclusion = document.getElementById("conclusion") ;
        maConclusion.style.color = "#006600" ;
        maConclusion.style.paddingLeft = "30px" ;
        maConclusion.style.fontVariant = "small-caps" ;
    </script>
</body>
</html>

```

Analysons ce script :

- Dans la variable maConclusion, nous référençons l'élément HTML à cibler.
- Avec le sous-objet style et avec la propriété color, nous modifions la couleur du texte existante.
- Avec le sous-objet style et avec la propriété paddingLeft, nous modifions le remplissage gauche existant.
- Avec le sous-objet style et avec la propriété fontVariant, nous ajoutons un nouveau style.

Voici l'affichage obtenu :



Mais ce qui importe le plus, c'est le code généré par cette modification :

```

<p id="conclusion" class="texte-important"
style="color: rgb(0, 102, 0); padding-left: 30px;
font-variant: small-caps;">Integer posuere erat a ante venenatis
dapibus posuere...</p>

```

Nous voyons que les deux styles existants dans la règle CSS initiale, color et paddingLeft, ont bien été modifiés. Ces modifications ont été ajoutées dans l'attribut style de l'élément ciblé. Comme vous le savez, les styles en ligne sont prioritaires par rapport aux styles déclarés dans la page .html et dans un fichier .css. C'est donc en toute logique que ces styles sont bien utilisés dans l'affichage de cet élément.

Enfin, le nouveau style CSS fontVariant est lui aussi ajouté en un style en ligne, comme nous l'avons vu précédemment.

L'exemple à télécharger est dans le dossier **Chapitre-05-C/exemple-03.html**.

Affichage dynamique dans un formulaire

Dans cet exemple, nous allons donner la possibilité aux utilisateurs d'afficher et de masquer une partie d'un formulaire. C'est une fonctionnalité assez classique qui permet d'afficher des options quand l'utilisateur clique sur une case à cocher, par exemple.

Voici la structure de la page HTML et du formulaire :

```
<form id="inscription" name="formulaire" method="post" action="#">
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <form id="inscription" name="formulaire" method="post" action="#">
      <p>
        <label for="nom">Nom : </label>
        <input id="nom" name="nom" type="text">
      </p>
      <p>
        <label for="prenom">Prénom : </label>
        <input id="prenom" name="prenom" type="text">
      </p>
      <p>Voir les options d'inscription : <input type="checkbox"
name="caseOptions" id="caseOptions"></p>
      <div id="divOptions">
        <fieldset>
          <legend>Les options d'inscription</legend>
          <p>
            <label for="option1">Option 1</label>
            <input id="option1" name="option1" type="checkbox">
          </p>
          <p>
            <label for="option2">Option 2</label>
            <input id="option" name="option2" type="checkbox">
          </p>
          <p>
            <label for="option3">Option 3</label>
            <input id="option3" name="option3" type="checkbox">
          </p>
        </fieldset>
      </div>
      <p>
        <input type="submit" name="envoyer" id="envoyer"
value="Envoyer">
      </p>
    </form>
  </body>
</html>
```

Nous avons une structure très classique. Ce qui nous intéresse dans un premier temps, c'est la case à cocher dont l'identifiant est caseOptions :

```
<input type="checkbox" name="caseOptions" id="caseOptions">
```

C'est cette case à cocher qui sera utilisée pour afficher ou masquer les options supplémentaires du formulaire.

Toutes les options supplémentaires sont insérées dans une boîte <div> dont l'identifiant est divOptions. C'est cette boîte qui sera masquée initialement et qui sera affichée par l'utilisateur :

```
<div id="divOptions">
    ...
</div>
```

Voilà le script qui est utilisé :

```
<script>
    document.getElementById("divOptions").style.display = "none" ;
    document.getElementById("caseOptions").onclick = function () {
        if (document.getElementById("caseOptions").checked) {
            document.getElementById("divOptions").style.display = "block" ;
        } else {
            document.getElementById("divOptions").style.display = "none" ;
        }
    }
</script>
```

Étudions ce script :

La première ligne permet de masquer les options supplémentaires, dès le chargement de la page :

- Nous référençons la boîte <div> des options, identifiée par divOptions.
- Et nous lui appliquons la propriété CSS display à none pour ne pas l'afficher.

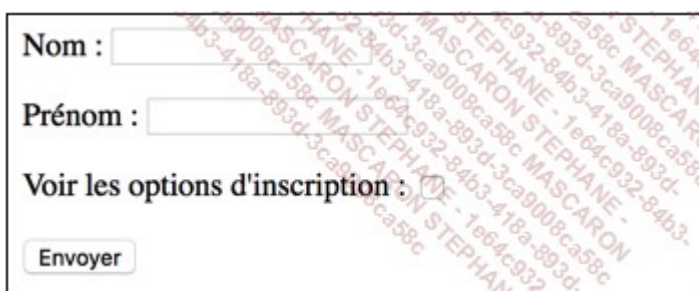
La deuxième ligne permet d'associer une fonction anonyme à la case à cocher :

- Nous référençons la case à cocher identifiée par caseOptions.
- Sur l'événement onclick, nous lui associons une fonction anonyme, function(){}.

Ensuite, nous devons effectuer un test pour savoir si la case est cochée ou non :

- Dans le if(), nous référençons la case à cocher caseOptions et nous testons si elle est cochée avec checked.
- Si c'est le cas, nous référençons la boîte <div> des options supplémentaires et lui appliquons la propriété CSS display à block pour l'afficher.
- Si ce n'est pas le cas, si elle n'est pas cochée, nous appliquons la propriété CSS display à none pour masquer la boîte <div> des options supplémentaires.

Voici l'affichage initial :



The screenshot shows a web form with the following elements: a label 'Nom :' followed by a text input field; a label 'Prénom :' followed by a text input field; a label 'Voir les options d'inscription :' followed by an unchecked checkbox; and a button labeled 'Envoyer' at the bottom left. The entire form is enclosed in a thin black border. A diagonal watermark is visible across the image, reading 'MASCARON - 1e64c932-84b3-418a-893d-3ca9008ca58c STEPHANE'.

Voici l'affichage obtenu lorsque la case est cochée :

Nom :

Prénom :

Voir les options d'inscription : ☒

Les options d'inscription

Option 1 ☐

Option 2 ☐

Option 3 ☐

L'exemple à télécharger est dans le dossier **Chapitre-05-D/exemple-01.html**.

Les bibliothèques JavaScript

Dans ce chapitre, nous allons aborder quelques bibliothèques JavaScript qui vont vous permettre de dynamiser vos pages web. Sachez qu'il existe des milliers de bibliothèques JavaScript. Certaines ne fonctionnent qu'en JavaScript "pur" et d'autres fonctionnent avec des frameworks JavaScript comme **jQuery**, qui est la plus célèbre. Dans les exemples qui suivent, nous n'allons étudier que des bibliothèques JavaScript sans dépendances.

Avec les connaissances que vous avez acquises dans les chapitres précédents, vous avez toutes les capacités nécessaires pour comprendre et personnaliser ces scripts.

Nous n'allons pas étudier chaque bibliothèque de manière exhaustive, en détaillant chaque option disponible. L'objectif est que vous compreniez comment ces bibliothèques fonctionnent et que vous puissiez mettre en œuvre rapidement ces scripts dans vos pages web. Ensuite, cela sera à vous d'approfondir le paramétrage de ces scripts en étudiant leur documentation, pour aller plus loin dans leur exploitation, pour les personnaliser à vos besoins. Puis, la dernière étape sera de créer vos propres scripts.

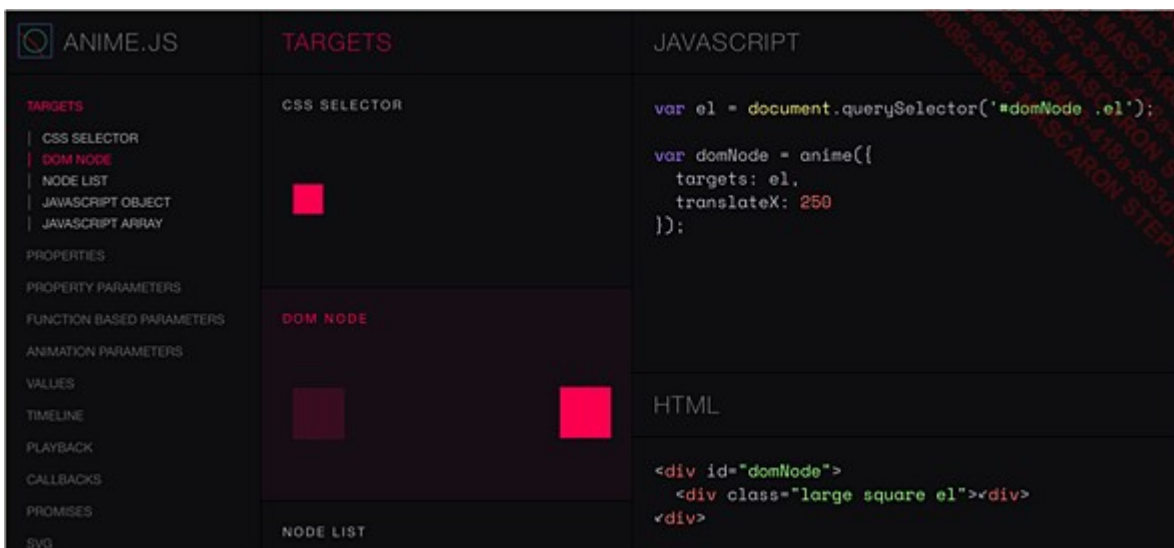
La bibliothèque Anime

1. Les objectifs

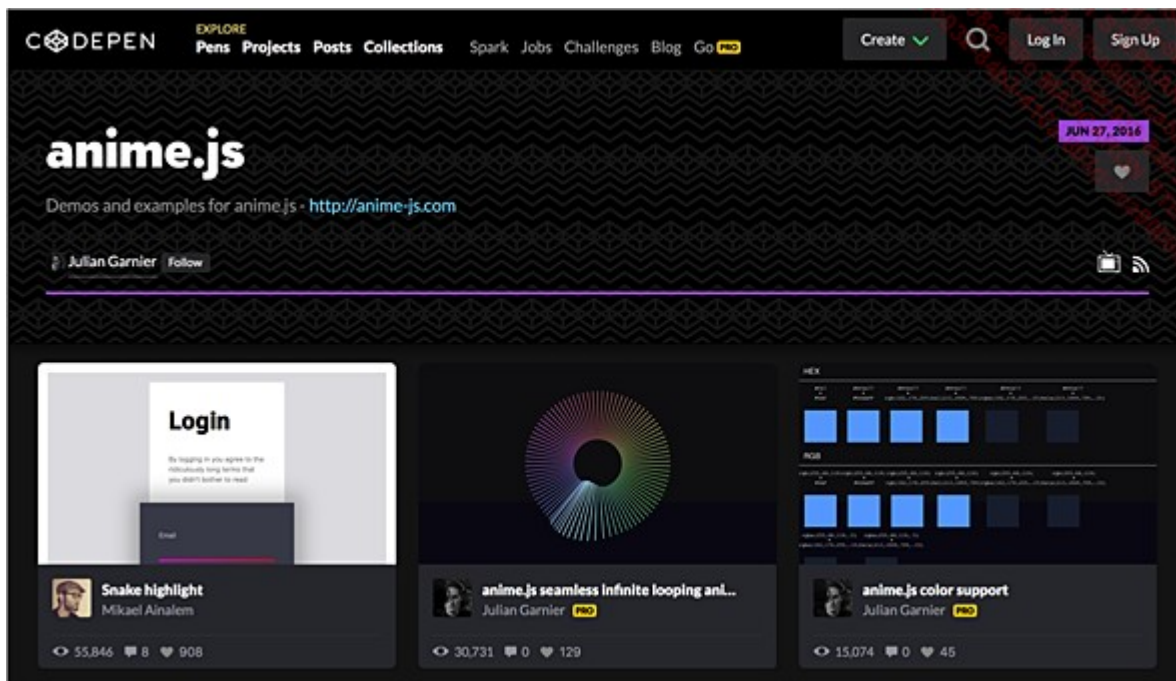
La bibliothèque **anime.js** (<http://animejs.com>) nous propose toute une série de scripts qui vont permettre d'animer des éléments des interfaces de nos pages web. Ces animations vont s'appliquer sur des propriétés CSS3, des transformations CSS3 et des animations CSS3.



Cette bibliothèque fournit les bases de la programmation à travers une documentation fournie, accessible par le bouton **Documentation** sur la page d'accueil (<http://animejs.com/documentation/>).



C'est ensuite aux développeurs de s'appuyer sur ces bases pour développer leurs propres applications, les animations qui répondent à leurs besoins. Nombre d'entre eux nous proposent leurs créations via le célèbre site de partage de code **codepen.io** (<https://codepen.io>). Ces créations sont accessibles via le bouton **Codepen** sur la page d'accueil (<https://codepen.io/collection/XLebem/>).



2. Installer la bibliothèque

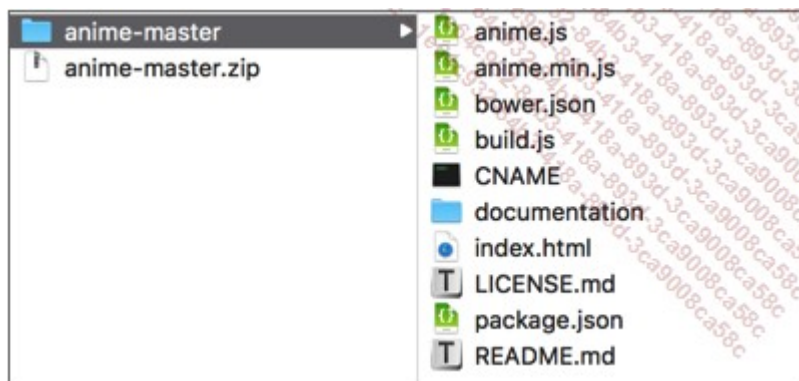
La première étape consiste à installer la bibliothèque et à la lier aux pages qui utiliseront cette bibliothèque.

Pour installer la bibliothèque **anime.js** dans vos pages web, sur la page d'accueil du site, cliquez sur le bouton **Github**.

Dans la zone **Usage**, cliquez sur le lien **download** pour télécharger la bibliothèque.

Vous téléchargez une archive nommée **anime-master.zip**.

Décompressez cette archive pour obtenir un dossier nommé **anime-master**.



Vous pouvez utiliser le fichier **anime.js** qui est non minimisé et qui pèse 31 ko pour étudier le code JavaScript. Le fichier **anime.min.js** est plutôt fait pour la production, puisqu'il est minimisé et qu'il ne pèse que 14 ko.

Placez le fichier **anime.min.js** dans le dossier de votre choix (**js**, par exemple), ou à la racine du dossier de votre site web.

Dans chaque fichier **.html**, vous devez lier cette bibliothèque pour exploiter ses fonctionnalités, avec l'élément HTML `<script>`.

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    ...
    <script src="anime.min.js"></script>
    ...
  </body>
</html>

```

3. Cibler les éléments à animer avec les CSS

Pour exploiter cette bibliothèque, il faut impérativement indiquer quel est l'élément à animer. Pour ce faire, cette bibliothèque nous propose plusieurs types de sélecteurs.

Le premier sélecteur que nous allons aborder est le sélecteur CSS. Nous allons cibler l'élément à animer par son sélecteur CSS qui peut être, par exemple, son identifiant ou sa classe.

L'élément à animer doit se placer dans un contexte d'animation, dans un conteneur. Dans cet exemple, c'est la boîte `<div id="contexte">`. Dans cette boîte, nous avons l'élément à animer, qui est aussi une boîte `<div>` qui utilise la classe `carre_rouge`. Cette classe est uniquement dédiée à l'affichage de l'élément à animer.

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .carre_rouge {
        background-color: #FF0000 ;
        width: 100px;
        height: 100px;
      }
    </style>
  </head>
  <body>
    <div id="contexte">
      <div class="carre_rouge"></div>
    </div>
    <script src="anime.min.js"></script>
  </body>
</html>

```

Après la liaison à la bibliothèque, nous avons le script d'animation. Ce script déplace vers la droite l'élément ciblé.

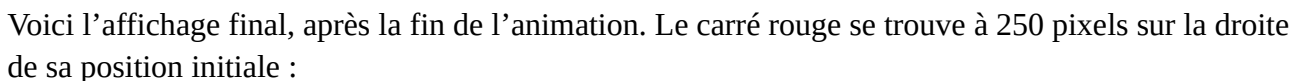
```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .carre_rouge {
        background-color: #FF0000 ;
        width: 100px;
        height: 100px;
      }
    </style>
  </head>
  <body>
    <div id="contexte">
      <div class="carre_rouge"></div>
    </div>
    <script src="anime.min.js"></script>
  </body>
</html>

```

Étudions ce script simple :

- Voici l’affichage initial de la page avant le déclenchement de l’animation. Le carré rouge se trouve sur la gauche de la page :



```
<!DOCTYPE html>
```

```

<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .carre_rouge {
        background-color: #FF0000 ;
        width: 100px;
        height: 100px;
      }
    </style>
  </head>
  <body>
    <div id="contexte">
      <div class="carre_rouge"></div>
    </div>
    <script src="anime.min.js"></script>
  </body>
</html>

```

Voici le script utilisé :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .carre_rouge {
        background-color: #FF0000 ;
        width: 100px;
        height: 100px;
      }
    </style>
  </head>
  <body>
    <div id="contexte">
      <div class="carre_rouge"></div>
    </div>
    <script src="anime.min.js"></script>
    <script>
      var elements_animes =
document.querySelector('#contexte .carre_rouge');
      var anime_carre_rouge = anime({
        targets: elements_animes,
        translateX: 250
      });
    </script>
  </body>
</html>

```

Analysons ce code simple :

- La variable nommée `elements_animes` permet de sélectionner les deux éléments intervenant dans l'animation : `#contexte` et `.carre_rouge`.
- La variable `anime_carre_rouge` utilise la fonction native `anime()`, vue précédemment. Les deux arguments sont les mêmes : les éléments de l'animation, `elements_animes`, et l'animation `translateX`.

Les affichages sont strictement identiques à l'exemple précédent.

L'exemple à télécharger est dans le dossier **Chapitre-06-B/exemple-02.html**.

5. Les animations avec des propriétés CSS

Dans cet exemple, nous allons utiliser quelques propriétés CSS pour animer un élément ciblé.

```
<div id="contexte">
  <div class="carre_rouge"></div>
</div>
```

Voici le script utilisé :

```
<script>
  var anime_CSS = anime({
    targets: '#contexte .carre_rouge',
    opacity: .5,
    backgroundColor: '#030',
    borderRadius: ['0em', '3em'],
    easing: 'easeInOutQuad'
  });
</script>
```

Étudions ce script :

- La variable nommée anime_CSS utilise la fonction native anime().
- Le premier argument, comme précédemment, cible les éléments concernés par l'animation.
- Les arguments suivants sont des propriétés CSS auxquelles nous attribuons des valeurs, pour obtenir une animation :
 - opacity : pour appliquer une transparence.
 - backgroundColor : pour appliquer une couleur d'arrière-plan.
 - borderRadius : pour appliquer des coins arrondis.
 - easing : pour appliquer une vitesse d'exécution.



Attention, souvenez-vous que JavaScript n'autorise pas le caractère - dans le nom des objets. C'est pour cela que, par exemple, la propriété CSS background-color est renommée backgroundColor.

Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .carre_rouge {
        background-color: #FF0000 ;
        width: 100px;
        height: 100px;
      }
      #contexte {
        margin-top: 50px;
      }
    </style>
  </head>
  <body>
    <div id="contexte">
      <div class="carre_rouge"></div>
    </div>
  </body>
</html>
```



```

    </style>
</head>
<body>
  <div id="contexte">
    <div class="carre_rouge"></div>
  </div>
  <script src="anime.min.js"></script>
  <script>
    var anime_CSS = anime({
      targets: '#contexte .carre_rouge',
      opacity: .5,
      backgroundColor: '#030',
      borderRadius: ['0em', '3em'],
      easing: 'easeInOutQuad'
    });
  </script>
</body>
</html>

```

Voici l’affichage initial de la page avant le déclenchement de l’animation. Le carré rouge se trouve sur la gauche de la page :



Voici l’affichage final obtenu. Le carré rouge est transformé en un rond vert :



L’exemple à télécharger est dans le dossier **Chapitre-06-B/exemple-03.html**.

6. Les animations avec des transformations CSS

Dans cet exemple, nous allons utiliser quelques propriétés CSS pour transformer un élément ciblé. Nous reprenons toujours le même contexte.

Voici le script utilisé :

```

<script>
  var transforme_carre_rouge = anime({

```

```

        targets: '#contexte .carre_rouge',
        translateX: 250,
        scale: 2,
        rotate: '1turn',
        duration: 5000
    });
</script>

```

Étudions ce script simple :

- La variable `transforme_carre_rouge` utilise la fonction native `anime()`.
- Les cibles sont les mêmes que précédemment.
- Nous effectuons un déplacement horizontal vers la droite de 250 pixels : `translateX: 250`.
- Nous doublons la taille du carré rouge : `scale: 2`.
- Nous effectuons une rotation complète d'un tour : `rotate: '1turn'`.
- Enfin, l'animation dure cinq secondes : `duration: 5000`.

Voici le code complet de cet exemple :

```

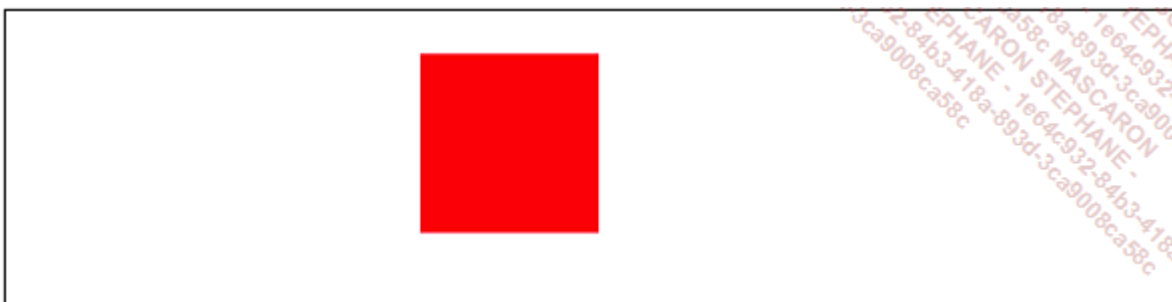
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .carre_rouge {
        background-color: #FF0000 ;
        width: 50px;
        height: 50px;
      }
      #contexte {
        margin-top: 50px;
      }
    </style>
  </head>
  <body>
    <div id="contexte">
      <div class="carre_rouge"></div>
    </div>
    <script src="anime.min.js"></script>
    <script>
      var transforme_carre_rouge = anime({
        targets: '#contexte .carre_rouge',
        translateX: 250,
        scale: 2,
        rotate: '1turn',
        duration: 5000
      });
    </script>
  </body>
</html>

```

Voici l'affichage initial de la page avant le déclenchement de l'animation. Le carré rouge se trouve sur la gauche de la page :



Voici l’affichage obtenu à la fin de l’animation :

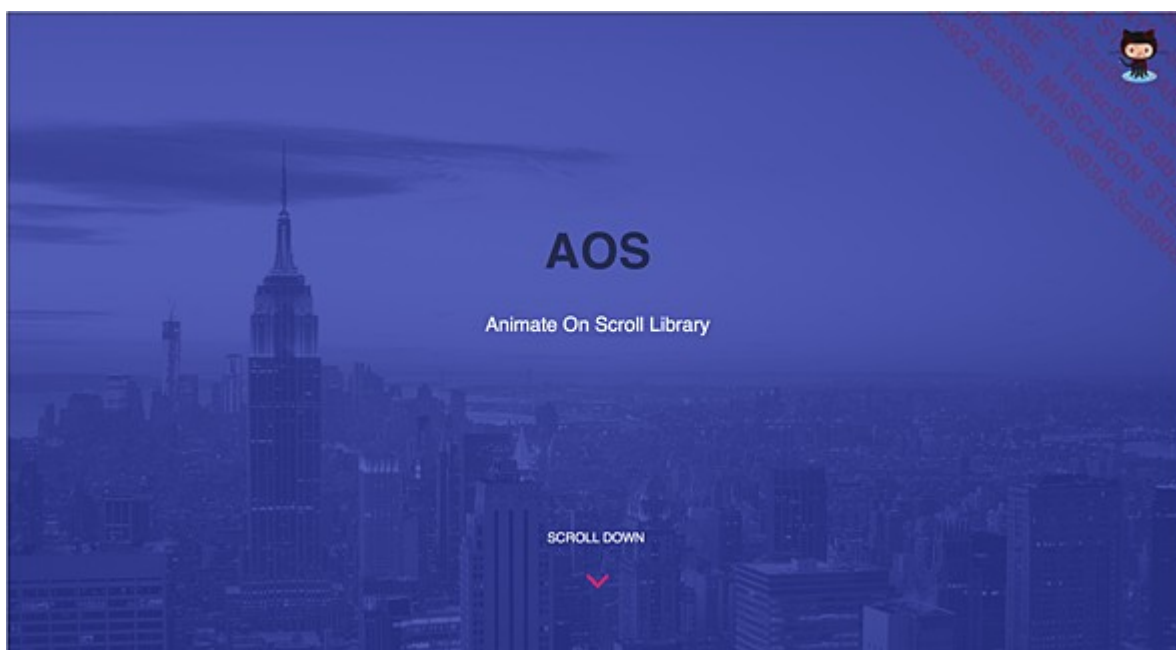


L’exemple à télécharger est dans le dossier **Chapitre-06-B/exemple-04.html**.

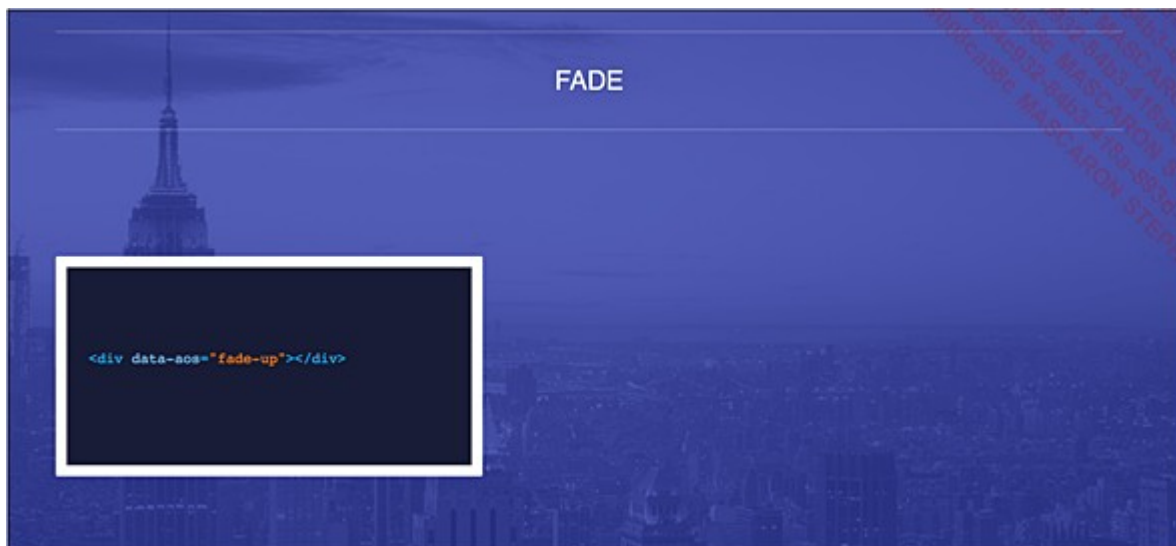
La bibliothèque Animate On Scroll

1. Les objectifs

La bibliothèque **Animate On Scroll** (<https://michalsnik.github.io/aos/>) va permettre d’afficher des éléments HTML, des boîtes <div> contenant des images, du texte..., avec des effets d’apparition divers, lorsque ces éléments arriveront à l’écran lors de l’utilisation de la barre de défilement vertical.



De nombreux exemples vous sont proposés sur la page d’accueil du site et ils apparaîtront en utilisant la barre de défilement vertical :



Toute la documentation technique se trouve sur GitHub : <https://github.com/michalsnik/aos>

2. Installer la bibliothèque

Pour exploiter cette bibliothèque, vous devez bien sûr l'installer et l'initialiser :

```
<script src="https://cdn.rawgit.com/michalsnik/aos/2.1.1/dist/aos.js"></script>
<script>
  AOS.init();
</script>
```

Vous devez aussi utiliser ses règles CSS, accessibles depuis un CDN :

```
<link href="https://cdn.rawgit.com/michalsnik/aos/2.1.1/dist/aos.css"
rel="stylesheet">
```

Voici la page complète avec les liaisons CSS et JS, sans aucune animation :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <link href="https://cdn.rawgit.com/michalsnik/aos/2.1.1/dist/aos.css"
rel="stylesheet">
  </head>
  <body>
    <script src="https://cdn.rawgit.com/michalsnik/aos/2.1.1/dist/aos.js">
</script>
    <script>
      AOS.init();
    </script>
  </body>
</html>
```

3. Les éléments d'animation

Dans cet exemple, nous allons animer des boîtes <div> contenant la même image :

```
<div class="photo">
  
</div>
```

Toutes ces boîtes utiliseront la classe `.photo`, afin de gérer un affichage commun :

```
<style>
  .photo {
    border: 10px solid #333;
    width: 260px;
    height: 173px;
  }
</style>
```

Afin de pouvoir utiliser la barre de défilement, les boîtes des images seront insérées dans d'autres boîtes qui occuperont toute la largeur de l'écran et une hauteur importante :

```
<div class="conteneur">
  <div class="photo">
    
  </div>
</div>
```

Voici les règles CSS de cette boîte :

```
.conteneur {
  width: 100%;
  height: 500px;
  float: left;
}
```

4. Les paramètres d'animation

C'est dans la boîte `<div class="photo">`, qui contient la photo, que nous devons paramétrer l'animation.

Le premier paramètre est `data-aos` et il indique quelle est l'animation à utiliser. Il en existe de très nombreuses :

- les fondus : `fade`, `fade-up`, `fade-down`, `fade-left`...
- les rotations : `flip-up`, `flip-down`...
- les apparitions : `slide-up`, `slide-down`...
- les effets de zoom : `zoom-in`, `zoom-in-down`, `zoom-in-left`, `zoom-out`...

Voici un exemple de syntaxe :

```
<div class="photo" data-aos="fade-zoom-in">
  
</div>
```

L'attribut `data-aos-duration` permet d'indiquer la durée de l'animation en millisecondes.

Voici un exemple de syntaxe :

```
<div class="photo" data-aos="fade-zoom-in" data-aos-duration="2000">
  
</div>
```

L'attribut `data-aos-easing` permet d'indiquer la vitesse d'exécution de l'animation. Voici une URL où vous trouverez des indications techniques sur les différentes courbes d'exécution des animations : <https://easings.net/fr>

Voici un exemple de syntaxe :

```
<div class="photo" data-aos="fade-zoom-in" data-aos-duration="2000" data-aos-easing="ease-in-sine">
  
</div>
```

L'attribut data-aos-offset permet d'indiquer une distance supplémentaire en pixels par rapport à l'apparition de l'élément à animer dans la page. C'est cette distance supplémentaire qui va déclencher l'animation.

Voici un exemple de syntaxe :

```
<div class="photo" data-aos="fade-zoom-in" data-aos-duration="2000" data-aos-easing="ease-in-sine" data-aos-offset="200">
  
</div>
```

5. Le code complet de cet exemple

Voici le code complet de cet exemple, avec ces quatre animations :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <link href="https://cdn.rawgit.com/michalsnik/aos/2.1.1/dist/aos.css" rel="stylesheet">
    <style>
      .conteneur {
        width: 100%;
        height: 500px;
        float: left;
      }
      .photo {
        border: 10px solid #333;
        width: 260px;
        height: 173px;
      }
    </style>
  </head>
  <body>
    <div class="conteneur">
      <div class="photo" data-aos="fade-zoom-in" data-aos-offset="200" data-aos-easing="ease-in-sine" data-aos-duration="2000">
        
      </div>
    </div>
    <div class="conteneur">
      <div class="photo" data-aos="fade-right" data-aos-offset="200" data-aos-easing="ease-in-sine" data-aos-duration="2000">
        
      </div>
    </div>
    <div class="conteneur">
      <div class="photo" data-aos="flip-right" data-aos-offset="200" data-aos-easing="ease-in-sine" data-aos-duration="2000">
        
      </div>
    </div>
  </body>
</html>
```

```

        <div class="photo" data-aos="zoom-in" data-aos-offset="200"
data-aos-easing="ease-in-sine" data-aos-duration="2000">
          
        </div>
      </div>
      <script
src="https://cdn.rawgit.com/michalsnik/aos/2.1.1/dist/aos.js"></script>
      <script>
        AOS.init();
      </script>
    </body>
  </html>

```

L'exemple à télécharger est dans le dossier **Chapitre-06-C/exemple-01.html**.

La bibliothèque Cleave

1. Les objectifs

La bibliothèque **Cleave** (<http://nosir.github.io/cleave.js/>) permet d'effectuer des vérifications et des validations de motifs de saisie dans les formulaires. Les motifs de saisie, patterns en anglais, sont des règles qui concernent les caractères à saisir ou à ne pas saisir, que doivent respecter les champs de saisie.



Les champs utilisables sont de nombreux types et il existe de nombreuses options pour chaque type de champ. Vous avez à votre disposition une documentation détaillée sur le GitHub de cette bibliothèque : <https://github.com/nosir/cleave.js>

2. Installer la bibliothèque

Pour installer **Cleave**, vous pouvez télécharger le fichier **cleave-react.min.js** minimisé à cette URL : <https://github.com/nosir/cleave.js/tree/master/dist>

Dans le dossier de votre site web, placez ce fichier où cela vous semble le mieux. Dans cet exemple simple, le fichier est à la racine du dossier du site.

Ensuite, vous faites une liaison classique au fichier **.js** :


```
<script src="cleave.min.js"></script>
```

Voici la structure de la page qui va servir dans l'exemple qui va suivre :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <script src="cleave.min.js"></script>
  </body>
</html>
```

3. Le champ de saisie de la date

Dans cet exemple simple, nous allons utiliser la bibliothèque pour trois types de champ de saisie :

- saisie d'une date,
- saisie d'un nombre,
- saisie d'un code composé de caractères devant répondre à un motif précis.

Le premier champ de saisie utilisé est un champ dans lequel l'utilisateur doit saisir une date avec un format précis : l'année en quatre chiffres, le mois et le jour en deux chiffres.

Voici le champ dans le formulaire qui utilise la classe `.input-date` :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <form method="post" action="#">
      <p>
        <label for="date">Saisissez une date : </label>
        <input id="date" class="input-date" type="text"/>
      </p>
    </form>
    <script src="cleave.min.js"></script>
  </body>
</html>
```

Après la liaison au fichier JavaScript de Cleave, nous insérons le script pour paramétrer cette saisie :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <form method="post" action="#">
      <p>
        <label for="date">Saisissez une date : </label>
        <input id="date" class="input-date" type="text"/>
      </p>
    </form>
  </body>
</html>
```

```


</form>
<script src="cleave.min.js"></script>
<script>
    var cleave = new Cleave('.input-date', {
        date: true,
        datePattern: ['d', 'm', 'Y']
    });
</script>
</body>
</html>

```

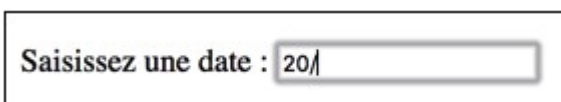
Analysons ce script :

- La variable nommée cleave utilise le constructeur new pour créer un nouvel objet Cleave().
- Le premier argument est l'identifiant du champ ciblé se faisant par l'utilisation de la classe .input-date.
- Le deuxième argument indique les options de validation de ce champ :
 - L'option date: true permet d'indiquer que le champ de texte doit recevoir une date.
 - L'option datePattern: précise entre crochets le motif (le pattern en anglais) de la saisie de la date. 'd' et 'm' indiquent que le jour et le mois doivent être saisis et affichés sur deux chiffres. 'Y' indique que l'année doit être saisie et affichée sur quatre chiffres.

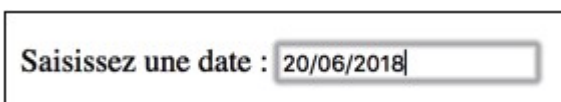
Voici l'affichage initial obtenu :



Dès que l'utilisateur commence à saisir la date, le script vérifie et formate correctement les chiffres :



Voici la saisie complète :



L'exemple à télécharger est dans le dossier **Chapitre-06-D/exemple-01.html**.

4. Le champ de saisie du nombre

Le deuxième champ est un champ dans lequel l'utilisateur doit saisir une valeur numérique avec un séparateur des milliers et un séparateur décimal spécifiés.

Voici le nouveau champ pour y saisir un nombre, il utilise la classe .input-number :

```

<p>
    <label for="nombre">Saisissez un nombre : </label>
    <input id="nombre" class="input-number" type="text"/>
</p>

```

Voici le script pour ce nouveau champ :

```
var cleave = new Cleave('.input-number', {
  numeral: true,
  numeralDecimalScale: 2,
  numeralDecimalMark: ',',
  delimiter: ' '
});
```

Étudions ce script :

- La variable nommée cleave utilise le constructeur new pour créer un nouvel objet Cleave().
- Le premier argument est l'identification du champ ciblé qui se fait par l'utilisation de la classe .input-number.
- Le deuxième argument indique les options de validation de ce champ :
 - L'option numeral: true permet d'indiquer que le champ de texte doit recevoir une valeur numérique.
 - L'option numeralDecimalScale: 2 indique que le champ n'acceptera que deux décimales.
 - L'option numeralDecimalMark: ',' indique que le séparateur décimal est une virgule.
 - L'option delimiter: ' ' indique que le séparateur des milliers est un espace.

Voici la page complète :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <form method="post" action="#">
      <p>
        <label for="nombre">Saisissez un nombre : </label>
        <input id="nombre" class="input-number" type="text"/>
      </p>
    </form>
    <script src="cleave.min.js"></script>
    <script>
      var cleave = new Cleave('.input-number', {
        numeral: true,
        numeralDecimalScale: 2,
        numeralDecimalMark: ',',
        delimiter: ' '
      });
    </script>
  </body>
</html>
```

Voici une saisie complète :

Saisissez un nombre :

L'exemple à télécharger est dans le dossier **Chapitre-06-D/exemple-02.html**.

5. Le champ de saisie d'un code

Le troisième champ de saisie est un champ dans lequel l'utilisateur doit saisir un code parfaitement défini. C'est une série de plusieurs caractères séparés par un séparateur déterminé.

Voici le champ qui utilise la classe `.input-code` :

```
<p>
  <label for="code">Saisissez un code : </label>
  <input id="code" class="input-code" type="text"/>
</p>
```

Voici le script :

```
var cleave = new Cleave('.input-code', {
  delimiter: '.',
  blocks: [3, 4, 3, 2],
  uppercase: true
});
```

Étudions ce script :

- La variable nommée `cleave` utilise le constructeur `new` pour créer un nouvel objet `Cleave()`.
- Le premier argument est l'identification du champ ciblé qui se fait par l'utilisation de la classe `.input-code`.
- Le deuxième argument indique les options de validation de ce champ :
 - L'option `delimiter: '.'` indique que le délimiteur des séries est un point.
 - L'option `blocks: [3, 4, 3, 2]` indique le nombre de caractères par série, par bloc. La première série doit comporter trois caractères, la deuxième quatre caractères, la troisième trois caractères et la quatrième et dernière, deux caractères.
 - L'option `uppercase: true` permet de n'afficher que des majuscules quand l'utilisateur saisit des lettres.

Voici la page complète :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <form method="post" action="#">
      <p>
        <label for="code">Saisissez un code : </label>
        <input id="code" class="input-code" type="text"/>
      </p>
    </form>
    <script src="cleave.min.js"></script>
    <script>
      var cleave = new Cleave('.input-code', {
        delimiter: '.',
        blocks: [3, 4, 3, 2],
        uppercase: true
      });
    </script>
```

```
</body>
</html>
```

Voici l'exemple du champ avec une saisie ne comportant que des nombres :

Saisissez un code :

Voici l'exemple du champ avec une saisie ne comportant que des lettres :

Saisissez un code :

Voici l'exemple du champ avec une saisie comportant des nombres et des lettres :

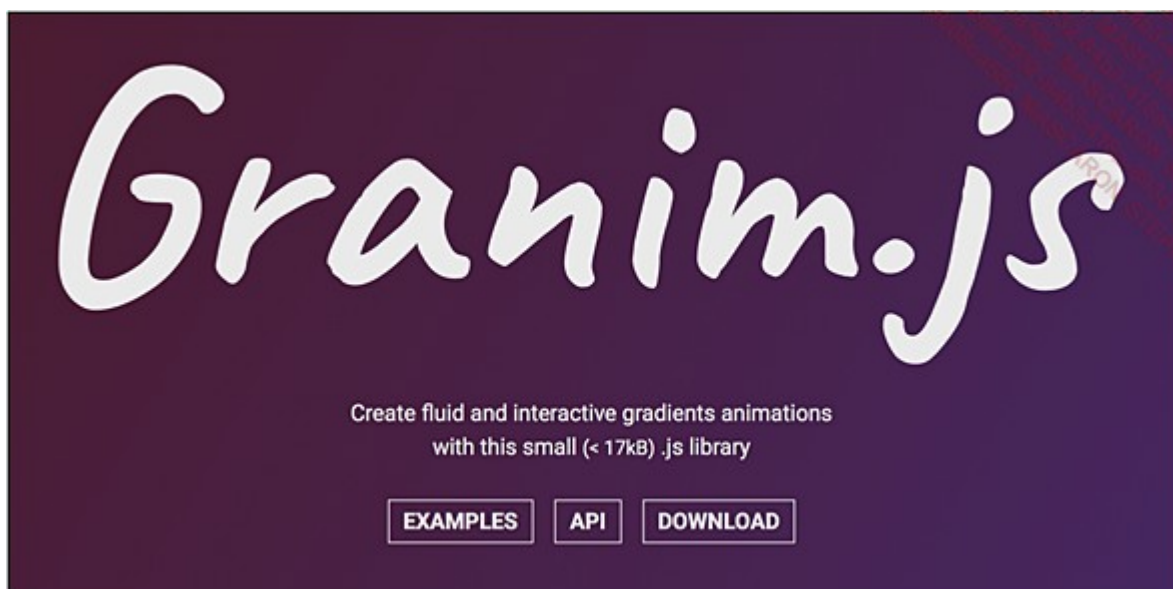
Saisissez un code :

L'exemple à télécharger est dans le dossier **Chapitre-06-D/exemple-03.html**.

La bibliothèque Granim

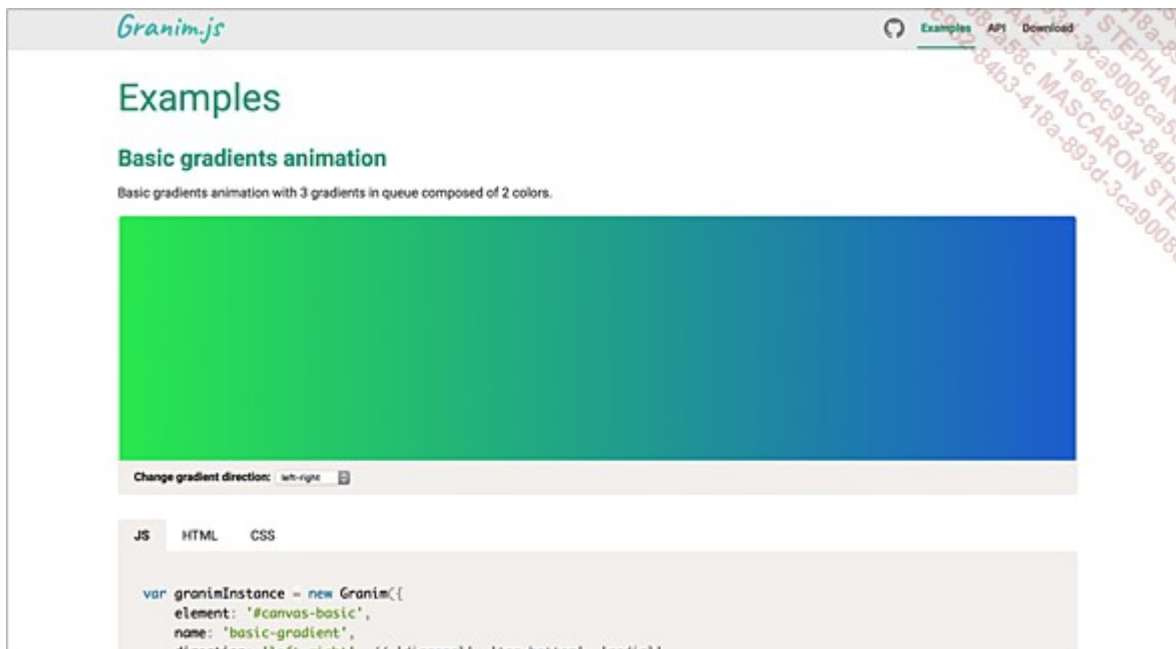
1. Les objectifs

La bibliothèque **Granim** (<https://sarcadass.github.io/granim.js/index.html>) permet de dynamiser vos pages web en animant des dégradés. Vous en avez la démonstration dès la page d'accueil du site.



Pour visualiser les différents effets des dégradés animés, cliquez sur le bouton

Exemples : <https://sarcadass.github.io/granim.js/examples.html>



Pour connaître tous les paramètres d'utilisation de cette bibliothèque, cliquez sur le bouton **API** (<https://sarcadass.github.io/granim.js/api.html>).

Property	Default	Type	Description
element (required)	/	String or HTMLCanvasElement	A CSS selector for the <code>canvas</code> element or the <code>HTMLCanvasElement</code> itself (e.g. <code>#granim-canvas</code> or <code>document.querySelector('#granim-canvas')</code>) that will be used for the gradient animation.
name	false	String	This is the prefix used for the dark / light class name added on the <code>options.elToSetClassOn</code> element depending on the average gradient lightness, the class will be updated during the animation. If you don't set a name, the class won't be set.
elToSetClassOn	'body'	String	The element to set the dark / light class on (e.g. <code>#canvas-wrapper</code>) Only useful if you set a name.
direction	'diagonal'	String	The orientation of the gradient, you can choose between: <ul style="list-style-type: none"> 'diagonal' 'left-right' 'top-bottom' 'radial'
isPausedWhenNotInView	false	Boolean	Does the animation stop when it's not in window view?

2. Installer la bibliothèque

Comme toujours, la première étape consiste à installer la bibliothèque dans vos pages web.

Dans la page d'accueil du site de Granim, cliquez sur le bouton **Download**.

Puis, sur la page du GitHub de Granim, téléchargez la dernière version disponible.

Vous téléchargez une archive nommée **granim.js-1.1.1.zip**. Le numéro est fonction de la version téléchargée, bien sûr.

Décompressez cette archive.

Dans le dossier **dist**, copiez le fichier JavaScript minimisé de Granim, **granim.min.js**.

Dans le dossier de votre site, placez ce fichier là où vous le souhaitez.

Dans chaque page HTML qui doit utiliser cette bibliothèque, faites une liaison au fichier JavaScript :

```
<script src="granim.min.js"></script>
```

Dans cet exemple, le fichier JavaScript est placé à la racine du dossier contenant le fichier HTML.

Voici la structure initiale de la page utilisée dans ces exemples :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <script src="granim.min.js"></script>
  </body>
</html>
```

3. Créer un dégradé animé avec trois teintes

Dans ce premier exemple, nous allons créer un dégradé qui va utiliser trois jeux de couleurs.



Attention, l'élément HTML qui va contenir le dégradé doit être un élément `<canvas>`.

Dans un premier temps, dans l'élément `<body>`, nous créons l'élément `<canvas>` qui va afficher le dégradé :

```
<canvas id="degrade" class="boite"></canvas>
```

Cet élément possède un identifiant qui sera utilisé dans le script. De plus, il possède aussi une classe CSS `.boite` (définie dans l'élément `<head>`) pour sa mise en page :

```
<style>
  .boite {
    width: 100%;
    height: 200px;
    float: left;
    border: 1px solid #333;
  }
</style>
```

La deuxième étape est la création du script qui va animer les dégradés dans l'élément :

```
<script>
  var degrade_teintes = new Granim({
    element: '#degrade',
    name: 'degrade',
    direction: 'left-right',
    opacity: [1, 1],
    isPausedWhenNotInView: true,
    states : {
      "default-state": {
        gradients: [
```



```

        ['#21a6ff', '#071daf'],
        ['#ffffb30', '#fc291e'],
        ['#1efc38', '#04560d']
    ]
}
});
</script>

```

Étudions ce script :

- La variable `degrade_teintes` utilise le constructeur `new` pour créer un nouvel objet `Granim()`.
- Nous avons ensuite les arguments du nouvel objet :
 - `element: '#degrade'` indique quel est l'élément HTML qui doit contenir le nouveau dégradé. Cet élément est référencé avec son identifiant `degrade`.
 - `name: 'degrade'` est le préfixe qui sera utilisé dans la définition de la classe de `Granim`.
 - `direction: 'left-right'` permet d'indiquer la direction des changements de teinte du dégradé. Vous pouvez utiliser `diagonal`, `left-right`, `top-bottom` et `radial`.
 - `opacity: [1, 1]` permet de spécifier la transparence des teintes.
 - `isPausedWhenNotInView: true` indique si l'animation doit être stoppée lorsque l'élément HTML n'est pas affiché dans la fenêtre du navigateur.
 - `states: {...}` contient tous les états du dégradé.
 - Le premier état est nommé par défaut `default-state{...}`.
 - `gradients: [...]` indique les différents dégradés utilisés. Le premier dégradé utilise les couleurs `#21a6ff` (bleu clair) et `#071daf` (bleu foncé). Le deuxième dégradé utilise les couleurs `#ffffb30` (jaune) et `#fc291e` (rouge). Et le troisième dégradé utilise les couleurs `#1efc38` (vert clair) et `#04560d` (vert foncé).

Bien sûr, il n'est pas possible d'avoir un rendu correct de ces dégradés colorés sur papier en noir et blanc, d'où l'absence de copies d'écran.

Voici le code complet de cet exemple :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .boite {
        width: 100%;
        height: 200px;
        float: left;
        border: 1px solid #333;
      }
    </style>
  </head>
  <body>
    <canvas id="degrade" class="boite"></canvas>
    <p>&nbsp;</p>
    <script src="granim.min.js"></script>
  </body>
</html>

```

```

<script>
    var degrade_teintes = new Granim({
        element: '#degrade',
        name: 'degrade',
        direction: 'left-right',
        opacity: [1, 1],
        isPausedWhenNotInView: true,
        states : {
            "default-state": {
                gradients: [
                    ['#21a6ff', '#071daf'],
                    ['#ffffb30', '#fc291e'],
                    ['#1efc38', '#04560d']
                ]
            }
        }
    });
</script>
</body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-06-E/exemple-01.html**.

4. Un dégradé animé au-dessus d'une image

La bibliothèque **Granim** nous permet d'appliquer un dégradé animé au-dessus d'une image. Vous obtiendrez un plus bel effet avec une image en noir et blanc. Nous reprenons la structure HTML, CSS et JavaScript de l'exemple précédent.

Voici l'élément HTML cible :

```
<canvas id="venise" class="boite"></canvas>
```

Voici le script associé :

```

var degrade_image = new Granim({
    element: '#venise',
    direction: 'top-bottom',
    opacity: [1, 1],
    isPausedWhenNotInView: true,
    image : {
        source: 'venise.jpg',
        blendingMode: 'multiply'
    },
    states : {
        "default-state": {
            gradients: [
                ['#29323c', '#485563'],
                ['#FF6B6B', '#556270'],
                ['#80d3fe', '#7ea0c4'],
                ['#f0ab51', '#eceba3']
            ],
            transitionSpeed: 7000
        }
    }
});

```

Analysons ce script :

- Nous retrouvons les éléments vus dans l'exemple précédent, nous n'y revenons pas.

- image: {...} permet d'indiquer quelle image doit être affichée dans l'élément HTML ciblé.
- source: 'venise.jpg' permet tout simplement d'indiquer le chemin d'accès à l'image.
- blendingMode: 'multiply' permet d'indiquer comment va se faire la fusion entre les couleurs de l'image et les couleurs des dégradés
(<https://developer.mozilla.org/fr/docs/Web/API/CanvasRenderingContext2D/globalCompositeOperation>).
- Comme précédemment, states: {...} contient tous les états du dégradé.
- transitionSpeed: 7000 indique la vitesse de transition entre les différents dégradés.

Voici l'affichage obtenu :



Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .boite {
        width: 100%;
        height: 200px;
        float: left;
        border: 1px solid #333;
      }
    </style>
  </head>
  <body>
    <canvas id="venise" class="boite"></canvas>
    <script src="granim.min.js"></script>
    <script>
      var degrade_image = new Granim({
        element: '#venise',
        direction: 'top-bottom',
        opacity: [1, 1],
        isPausedWhenNotInView: true,
        image : {
          source: 'venise.jpg',
          blendingMode: 'multiply'
        },
        states : {
          "default-state": {
            gradients: [
              ['#29323c', '#485563'],
              ['#FF6B6B', '#556270'],
              ['#80d3fe', '#7ea0c4'],
              ['#f0ab51', '#ecea3']
            ],
            transitionSpeed: 7000
          }
        }
      });
```

```
    </script>
  </body>
</html>
```

L'exemple à télécharger est dans le dossier **Chapitre-06-E/exemple-02.html**.

La bibliothèque Tippy

1. Les objectifs

La bibliothèque **Tippy** (<https://atomiks.github.io/tippyjs/>) permet d'insérer des bulles d'aide pouvant afficher du texte, des images ou tout autre élément HTML.



Pour fonctionner, **Tippy** s'appuie sur une autre bibliothèque JavaScript, **Popper** (<https://popper.js.org>). La documentation de Tippy se trouve sur GitHub : <https://github.com/atomiks/tippyjs>

2. Installer la bibliothèque Tippy

L'installation de Tippy se fait par une simple liaison vers un CDN, juste avant la balise de fermeture `</body>` :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    ...
    <script
src="https://unpkg.com/tippy.js/dist/tippy.all.min.js"></script>
  </body>
</html>
```



Notez que l'installation du fichier **.js** installe aussi les règles CSS utilisées par les bulles d'aide.

3. Ajouter une bulle d'aide textuelle simple

Dans ce premier exemple, nous allons ajouter une bulle d'aide à un texte. Le texte associé à la bulle d'aide devra pouvoir être référencé par un identifiant id. Le texte de la bulle d'aide est placé dans l'attribut title de ce texte.

Voici l'élément HTML utilisé :

```
<p>Le <span class="tippy" id="doges" title="Le palais des Doges ou palais Ducal est un palais vénitien de styles gothique et Renaissance situé sur la place Saint-Marc.">Palais des doges</span>.</p>
```

Voici ce code simple :

- Toute la phrase est insérée dans un simple élément `<p>`.
- Le texte activant la bulle d'aide est placé dans un élément ``.
- Cet élément utilise la classe `.tippy` pour sa mise en forme.
- Cet élément possède l'identifiant `id="doges"`.
- Cet élément utilise bien l'attribut `title`, avec le texte pour la bulle d'aide.

Voici la règle CSS `span.tippy`, placée dans l'élément `<head>` :

```
span.tippy {  
    text-decoration: underline dotted;  
}
```

Voici l'affichage initial :



Voyons maintenant le script :

```
<script>  
    tippy('#doges');  
</script>
```

Le script est extrêmement simple : la fonction native `tippy()` permet de créer un nouvel objet `Tippy`. Ce nouvel objet n'utilise ici qu'un seul argument, l'identifiant de l'élément où doit être affichée la bulle d'aide : `#doges`.

Dans cet exemple, la bulle d'aide utilise tous les paramètres d'affichage par défaut.

Voici l'affichage obtenu :



Ces paramètres par défaut incluent la position de la bulle d'aide. Celle-ci s'affiche là où il y a de la place. Dans l'exemple précédent, la bulle d'aide peut se placer au-dessus du texte concerné, car il y a de la place.

Voici un exemple où il n'y a pas assez de place pour afficher la bulle d'aide au-dessus du texte. Dans ce cas, la bulle d'aide s'affiche au-dessous :



Voici le code de ce premier exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      p {
        margin: 80px;
      }
      span.tippy {
        text-decoration: underline dotted;
      }
    </style>
  </head>
  <body>
    <br>
    <p>Le <span class="tippy" id="doges" title="Le palais des Doges ou
palais Ducal est un palais vénitien de styles gothique et Renaissance situé
sur la place Saint-Marc.">Palais des doges</span>.</p>
    <script src="https://unpkg.com/tippy.js/dist/tippy.all.min.js">
  </script>
    <script>
      tippy('#doges');
    </script>
  </body>
</html>
```

L'exemple à télécharger est dans le dossier **Chapitre-06-F/exemple-01.html**.

4. Ajouter une bulle d'aide textuelle paramétrée

Dans ce deuxième exemple, nous allons paramétrer l'affichage de la bulle d'aide. La structure HTML, CSS et JavaScript est similaire à l'exemple précédent.

Voici le texte concerné :

```
<p>La place <span class="tippy" id="sanmarco" title="La place Saint-Marc à
Venise">San Marco</span>, la place la plus visitée de Venise.</p>
```

Comme précédemment, le texte concerné utilise la classe .tippy, possède un identifiant id="sanmarco" et son attribut de titre title="La place Saint-Marc à Venise".

Voici son affichage initial :

La place San Marco, la place la plus visitée de Venise.

Voici le script associé :

```
<script>
  tippy(
    '#sanmarco',
    {
      arrow: true,
      arrowType: 'round',
      size: 'large',
      animation: 'fade',
      duration: [800, 200],
      distance: 20,
      followCursor: true
    }
  );
</script>
```

Les paramètres se placent en tant que deuxième argument de la fonction tippy(), séparé du premier par une virgule. Tous les paramètres du deuxième argument sont placés entre accolades {...} :
tippy('#sanmarco,{...})

Étudions les paramètres utilisés :

- arrow: true indique que nous souhaitons avoir une flèche pour la bulle d'aide.
- arrowType: 'round' précise que cette flèche doit avoir sa pointe arrondie.
- size: 'large' indique que la flèche doit avoir une large taille.
- animation: 'fade' signifie que la bulle d'aide doit apparaître avec un effet de fondu.
- duration: [800, 200] précise, avec la première valeur, la vitesse d'affichage de la bulle d'aide et, avec la deuxième valeur, la vitesse de fermeture, lorsque le pointeur de la souris sort de la zone d'affichage.
- distance: 20 indique la distance supplémentaire entre le texte et la bulle d'aide.
- followCursor: true signifie que la bulle d'aide doit suivre le déplacement du pointeur de la souris, au-dessus du texte ciblé.

Voici l'affichage obtenu lorsque le pointeur est sur la gauche du texte ciblé :



Voici l'affichage obtenu lorsque le pointeur est sur la droite du texte ciblé :



Voici le code complet pour ce deuxième exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      p {
        margin: 80px;
      }
      span.tippy {
        text-decoration: underline dotted;
      }
    </style>
  </head>
  <body>
    <br>
    <p>La place <span class="tippy" id="sanmarco" title="La place Saint-
Marc à Venise">San Marco</span>, la place la plus visitée de Venise.</p>
    <script src="https://unpkg.com/tippy.js/dist/tippy.all.min.js">
  </script>
    <script>
      tippy(
        '#sanmarco',
        {
          arrow: true,
          arrowType: 'round',
          size: 'large',
          animation: 'fade',
          duration: [800, 200],
          distance: 20,
          followCursor: true
        }
      );
    </script>
  </body>
</html>
```

L'exemple à télécharger est dans le dossier **Chapitre-06-F/exemple-02.html**.

5. Ajouter une bulle d'aide avec une image

Dans ce dernier exemple, nous allons ajouter une bulle d'aide qui contiendra un titre et une image.

Une des possibilités est de créer un élément HTML qui contient tous les éléments à afficher.

Dans cet exemple, c'est une simple boîte <div> qui possède bien sûr un identifiant, id="contenuSalute", et un style en ligne qui indique que cette boîte ne doit pas être affichée dans la page HTML : style="display: none;".

Voici le code très simple de cette boîte :

```
<div id="contenuSalute" style="display: none;">
  <h3>Santa Maria della Salute</h3>
  
</div>
```

Voici le paragraphe qui va permettre l'affichage de la bulle d'aide :

```
<p>La Basilique <span class="tippy" id="salute">Santa Maria della
Salute</span>.</p>
```

Nous y retrouvons tous les paramètres vus précédemment, nous n’y revenons pas.

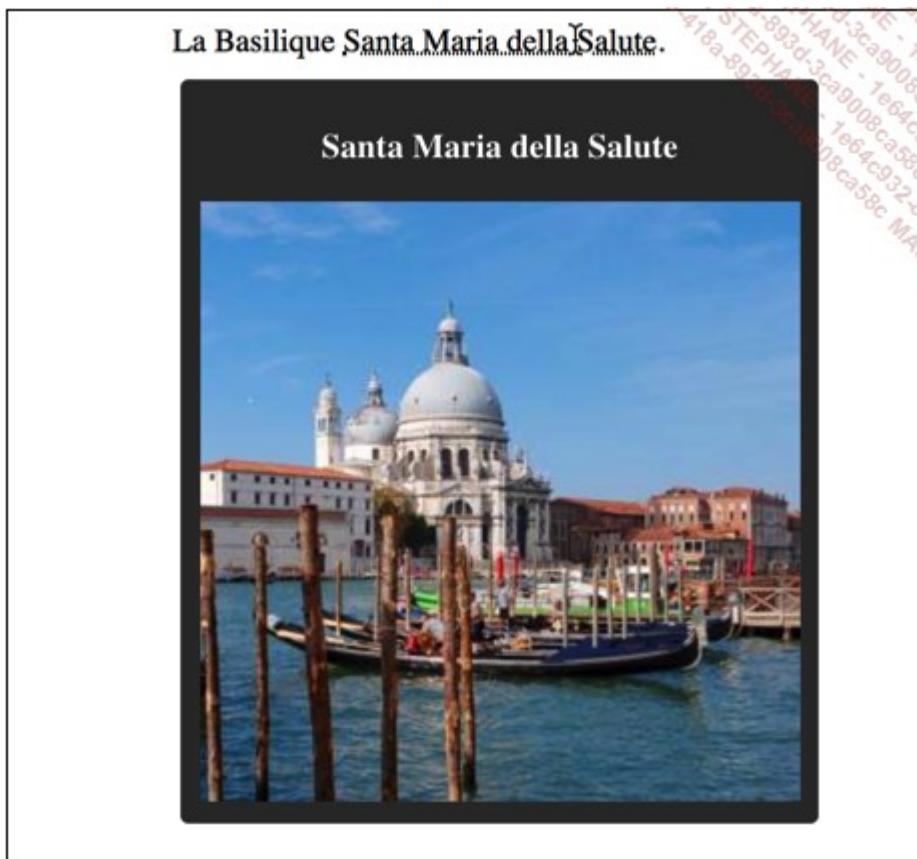
Voyons maintenant le script associé :

```
<script>
  tippy(
    '#salute',
    {
      html: '#contenuSalute'
    }
  );
</script>
```

Le nouvel objet tippy() utilise deux arguments :

- '#salute' précise quel est l'élément ciblé avec son identifiant.
- {html: '#contenuSalute'} indique quel élément HTML doit être injecté comme contenu de la bulle d'aide. La boîte <div> ciblée n'est toujours pas affichée dans la page HTML, mais elle est bien affichée comme contenu de la bulle d'aide.

Voici l’affichage obtenu :



Voici le code complet pour ce troisième exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      p {
        margin: 80px;
      }
    </style>
  </head>
  <body>
    <div>
      <a href="#">La Basilique Santa Maria della Salute</a>
    </div>
  </body>
</html>
```

```

        span.tippy {
            text-decoration: underline dotted;
        }
    </style>
</head>
<body>
    <p>La Basilique <span class="tippy" id="salute">Santa Maria della
Salute</span>.</p>
    <div id="contenuSalute" style="display: none;">
        <h3>Santa Maria della Salute</h3>
        
    </div>
    <script src="https://unpkg.com/tippy.js/dist/tippy.all.min.js">
</script>
    <script>
        tippy(
            '#salute',
            {
                html: '#contenuSalute'
            }
        );
    </script>
</body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-06-F/exemple-03.html**.

La bibliothèque Micron

1. Les objectifs

Cette très légère bibliothèque va permettre de dynamiser les boutons avec de petites animations. Voici son URL sur GitHub : <https://github.com/webkul/micron>



Sa documentation est aussi sur GitHub : <https://webkul.github.io/micron/docs.html>

2. L'installation de la bibliothèque

Pour installer cette bibliothèque, vous devez lier son fichier **.js** et son fichier **.css** à un CDN, dans l'élément **<head>** de vos fichiers **.html** :

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Ma page</title>
        <link
href="https://unpkg.com/webkul-micron@1.1.4/dist/css/micron.min.css"
type="text/css" rel="stylesheet">
        <script
src="https://unpkg.com/webkul-micron@1.1.4/dist/script/micron.min.js"
type="text/javascript"></script>
    </head>
    <body>
        ...

```

```
</body>
</html>
```

3. Insérer un bouton animé

Nous allons insérer un bouton avec l'élément `<button>` et avec la classe `.boutons` pour sa simple mise en forme :

```
<button class="boutons">Mon bouton</button>
```

Voici la classe `.boutons` :

```
<style>
  .boutons {
    width: 100px;
    height: 50px;
    box-shadow: 0px 0px 10px #343434;
    border-radius: 10px;
    border: 1px solid #656565;
    background-color: #eee;
  }
</style>
```

4. Animer le bouton

Pour animer le bouton, nous allons utiliser des attributs spécifiques à la bibliothèque.

L'attribut `data-micron` va permettre de spécifier l'animation voulue. Voici quelques effets disponibles :

- `data-micron="shake"` permet d'obtenir un effet de déplacement rapide de gauche à droite. Le bouton est "secoué" horizontalement.
- `data-micron="bounce"` permet d'obtenir un effet similaire, mais verticalement.
- `data-micron="fade"` permet d'obtenir un effet de fondu.
- `data-micron="jerk"` permet d'obtenir un effet de basculement horizontal.

Exemple avec l'effet `jerk` :

```
<button class="boutons" data-micron="jerk">Mon bouton</button>
```

L'attribut `data-micron-duration` permet d'indiquer la durée de l'animation. La valeur est exprimée en secondes.

Exemple avec une durée de 2 secondes :

```
<button class="boutons" data-micron="jerk"
data-micron-duration="2">Mon bouton</button>
```

L'attribut `data-micron-timing` permet de spécifier la vitesse d'exécution de l'animation. Vous pouvez choisir ces valeurs : `linear`, `ease-in`, `ease-out` et `ease-in-out`.

Exemple :

```
<p><button class="boutons" data-micron="jerk"
data-micron-duration="2" data-micron-timing="ease-in">Mon bouton
</button></p>
```

5. Le code complet de la page

Voici le code complet de la page, avec deux exemples de boutons animés :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .boutons {
        width: 100px;
        height: 50px;
        box-shadow: 0px 0px 10px #343434;
        border-radius: 10px;
        border: 1px solid #656565;
        background-color: #eee;
      }
    </style>
    <link
href="https://unpkg.com/webkul-micron@1.1.4/dist/css/micron.min.css"
type="text/css" rel="stylesheet">
    <script
src="https://unpkg.com/webkul-micron@1.1.4/dist/script/micron.min.js"
type="text/javascript"></script>
  </head>
  <body>
    <p><button class="boutons" data-micron="bounce"
data-micron-duration="1.5" data-micron-timing="ease-out">Mon
bouton</button></p>
    <p><button class="boutons" data-micron="jerk"
data-micron-duration="2" data-micron-timing="ease-in">Mon
bouton</button></p>
  </body>
</html>
```

L'exemple à télécharger est dans le dossier **Chapitre-06-G/exemple-01.html**.

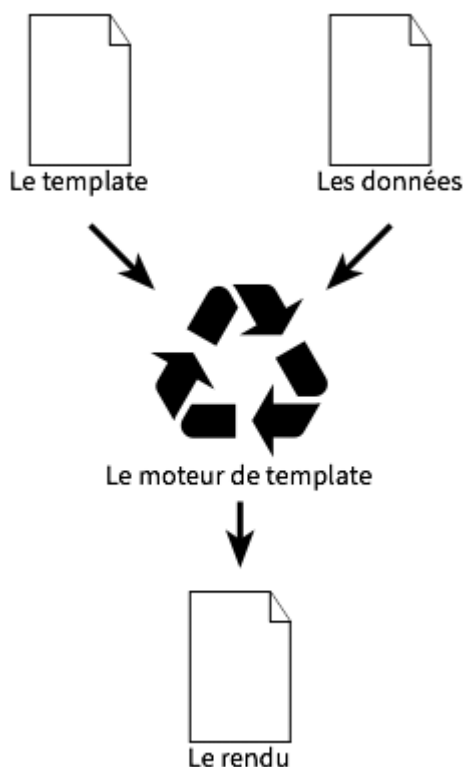
Présentation du moteur de rendu Mustache

Mustache est une spécification qui permet de créer des templates. Ce n'est pas un moteur de templates, c'est un moteur de rendu, un moteur de vues. Voici son URL : <http://mustache.github.io>

Autre point important, Mustache permet de créer des templates sans logique, il est dit logic-less en anglais. Cela veut dire que nous ne pourrons pas utiliser des fonctions de condition comme le `if()`, ni des boucles comme `while()` ou `for()`.

Pour concevoir nos pages, nous allons avoir besoin de trois composants :

- un template conçu avec les balises spécifiques de Mustache,
- des données qui peuvent être stockées dans le fichier `.html` ou à l'extérieur de celui-ci,
- un moteur de template.



Le premier composant est constitué par les balises. Avec les balises spécifiques de Mustache, nous allons concevoir des templates. Dans le template, nous indiquerons des emplacements qui seront ensuite remplacés par les données à afficher. La syntaxe de Mustache utilise des accolades `{{...}}` pour indiquer le nom des éléments qui seront ensuite remplacés par les données. Par exemple, si nous souhaitons afficher une destination de voyage provenant d'une liste, la syntaxe pourrait être la suivante : `{{destination}}`.

Le deuxième composant nécessaire est une source de données. Cette source de données peut être stockée dans le fichier **.html**, sous la forme d'un objet JavaScript que nous pourrons ensuite manipuler avec les méthodes et les propriétés des objets JavaScript. Les données peuvent être aussi stockées à l'extérieur du fichier **.html**. Ces données peuvent parfaitement provenir d'une autre source que d'une source interne à la page HTML. Elles peuvent provenir d'une base de données, d'un fichier **.txt**, d'un processus de création de fichier... Pour plus de facilité, pour pouvoir travailler

tout en JavaScript, ces données ont tout intérêt à être enregistrées au format JSON (JavaScript Object Notation), dans un fichier ayant l'extension **.json**.

Le troisième composant est le moteur de template, c'est lui qui va effectuer le rendu des données dans le template indiqué. Le moteur de rendu peut être utilisé avec de nombreux langages : JavaScript, PHP, Ruby, Python... Vous avez cette liste de langage indiquée sur la page d'accueil du site de Mustache.



À la fin de ce processus, nous obtiendrons un fichier HTML qui sera généré dynamiquement pour afficher les données dans le bon template.

Les données à afficher

1. La source des données

Les données doivent être disponibles dans un format qui soit en adéquation avec le moteur de rendu choisi. Si nous choisissons un moteur de rendu en JavaScript, les données pourront être aussi au format JavaScript. Nous avons deux possibilités :

- Les données pourront être des objets JavaScript qui seront insérés dans le fichier **.html** ou **.js**.
- Les données pourront être au format JSON et être placées dans un fichier **.json** externe.

2. Le format JSON

Le premier format des sources de données que nous pouvons utiliser est le format **JSON**. C'est un format qui a été créé par **Douglas Crockford** à partir de 2002. Il permet de stocker facilement et rapidement des données aisément accessibles par de très nombreux langages. C'est donc un format d'échange de données avant tout.

Le format JSON comporte deux composants : un nom (ou une clé) et une valeur. Les données peuvent être de trois types seulement :

- des objets,
- des tableaux,

- des valeurs de type objet, tableau, booléen, nombre, chaîne de caractères ou null.

Le nom des données et les valeurs, sauf les valeurs numériques, sont entre guillemets "..." et sont séparés par le caractère deux-points :.



Attention, vous ne pouvez pas utiliser des simples quotes '...'.

Toutes les données sont placées entre accolades et les couples nom/valeur sont séparés par une virgule ,. Pour le dernier couple, la virgule n'est pas requise.

Voici un exemple simple :

```
{
  "prenom": "Christophe",
  "nom": "Aubry",
  "adresse": {
    "numero": 12,
    "rue": "avenue des Plantes",
    "ville": "Nantes"
  }
}
```

Analysons cette structure simple :

- Toutes les données sont placées dans un couple d'accolades.
- Les deux premières lignes sont des couples nom/valeur classiques. Le nom et la valeur sont séparés par le caractère deux-points :. Les contenus des noms et des valeurs (sauf les valeurs numériques) sont placés entre guillemets "...".
- La troisième ligne est une donnée imbriquée de type objet qui possède ses propres couples nom/valeur. Ces données imbriquées sont donc elles aussi placées entre accolades.

Voici un autre exemple avec une donnée sous forme de tableau et une autre avec un booléen :

```
{
  "fruits": [
    {
      "Pommes": 3,
      "Ananas": 4,
      "Poires": 5
    },
    {
      "fraise": 6,
      "cerise": 7
    }
  ]
  "legume": false
}
```

La donnée nommée fruits est un tableau composé de deux objets. Le tableau est placé entre crochets [...]. La donnée legume est de type booléen, avec la valeur false.

Pour un contenu d'un site web, nous pourrions avoir cette première syntaxe avec des objets :

```
{
  "article1": {
    "titre": "Lorem Ipsum",
```

```

        "contenu": "Aenean lacinia bibendum nulla sed consectetur...",
        "date": "08/06/2018",
        "auteur": "Christophe"
    },
    "article2": {
        "titre": "Cursus Malesuada Amet",
        "contenu": "Praesent commodo cursus magna...",
        "date": "11/06/2018",
        "auteur": "Christine"
    },
    "article3": {
        "titre": "Maecenas faucibus",
        "contenu": "Vestibulum id ligula porta felis euismod...",
        "date": "18/06/2018",
        "auteur": "Mathis"
    },
    ...
}

```

Étudions cette structure :

- Nous avons trois objets, nommés "article1", "article2" et "article3".
- Les données de chacun de ces trois objets sont placées dans des accolades {...}.
- Chacun de ces trois objets comporte des couples nom/valeur :
 - "titre": "...",
 - "contenu": "...",
 - "date": "...",
 - "auteur": "...",

Nous pouvons aussi avoir une deuxième structure, cette fois en tableau :

```

[
    {
        "titre": "Lorem Ipsum",
        "contenu": "Aenean lacinia bibendum nulla sed consectetur...",
        "date": "08/06/2018",
        "auteur": "Christophe"
    },
    {
        "titre": "Cursus Malesuada Amet",
        "contenu": "Praesent commodo cursus magna...",
        "date": "11/06/2018",
        "auteur": "Christine"
    },
    {
        "titre": "Maecenas faucibus",
        "contenu": "Vestibulum id ligula porta felis euismod...",
        "date": "18/06/2018",
        "auteur": "Mathis"
    },
    ...
]

```

Analysons cette structure simple :

- Toutes les données sont tout de suite insérées dans un tableau qui est délimité par un couple de crochets [...].

- Chaque article est un objet placé entre accolades {...}.
- Chaque article possède un titre, titre, un contenu, contenu, une date de publication, date, et un auteur, auteur.

3. Les objets JavaScript internes

Les données peuvent être utilisées comme des objets JavaScript, en utilisant un var très classique.

Voici un exemple très simple de données sur des chiens indiquées ici pour l'exemple au format JSON :

```
{
  "Chien": "Canis",
  "chien1": {
    "Nom": "Raja",
    "Couleur": "Brun",
    "Race": "Labrador"
  },
  "chien2": {
    "Nom": "Lord",
    "Couleur": "Beige",
    "Race": "Beagle"
  }
}
```

Dans un fichier HTML, dans un élément <script>, nous pourrions avoir cette syntaxe :

```
<script>
  var chiensJSON =
  '{"Chien": "Canis", "chien1": {"Nom": "Raja", "Couleur": "Brun", "Race": "Labrador"}, "chien2": {"Nom": "Lord", "Couleur": "Beige", "Race": "Beagle"}}';
</script>
```

La variable nommée chiensJSON stocke des données au format JSON.

Pour utiliser ces données, nous utilisons l'objet JSON

(https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/JSON) et sa méthode parse (https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/JSON/parse).

```
var lesChiens = JSON.parse(chiensJSON);
```

Ensuite, nous pouvons afficher une donnée dans la console :

```
console.log(lesChiens.chien2.Nom);
```

Voici l'affichage obtenu :



Voici le code complet de cet exemple :

```
<!DOCTYPE html>
```

```

<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <script>
      var chiensJSON =
'{"Chien": "Canis", "chien1": {"Nom": "Raja", "Couleur": "Brun", "Race":
"Labrador"}, "chien2": {"Nom": "Lord", "Couleur": "Beige", "Race": "Beagle"}}';
      var lesChiens = JSON.parse(chiensJSON);
      console.log(lesChiens.chien2.Nom);
    </script>
  </body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-07-B/exemple-01.html**.

4. Le fichier JSON externe

L'autre solution est donc de créer un fichier externe **.json** contenant toutes les données au format JSON.

Voici le fichier de données nommé **articles.json** de cet exemple :

```

{
  "article1": {
    "titre": "Lorem Ipsum",
    "contenu": "Aenean lacinia bibendum nulla...",
    "date": "08/06/2018",
    "auteur": "Christophe"
  },
  "article2": {
    "titre": "Cursus Malesuada Amet",
    "contenu": "Praesent commodo cursus magna...",
    "date": "11/06/2018",
    "auteur": "Christine"
  },
  "article3": {
    "titre": "Maecenas faucibus",
    "contenu": "Vestibulum id ligula porta...",
    "date": "18/06/2018",
    "auteur": "Mathis"
  }
}

```

Voici la structure du fichier **.html** :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <script>
      function articlesJSON() {
        var httpRequestArticles = new XMLHttpRequest();
        httpRequestArticles.open("GET", "articles.json", true);
        httpRequestArticles.setRequestHeader("Content-type",
"application/json");
        httpRequestArticles.onreadystatechange = function() {
          if (httpRequestArticles.readyState == 4 &&
httpRequestArticles.status == 200) {

```

```

        var donneesJSON =
JSON.parse(httpRequestArticles.responseText);
        console.log(donneesJSON);
    }
}
httpRequestArticles.send(null);
}
</script>
</head>
<body>
    <script type="text/javascript">
        articlesJSON();
    </script>
</body>
</html>

```



Attention, la méthode que nous allons appliquer utilise le protocole HTTP, donc si vous voulez tester cet exemple, vous devez mettre en place un serveur local, avec MAMP, WAMP, ou toute autre solution de votre choix selon votre plateforme.

Détaillons les scripts de ce fichier :

- Nous déclarons une fonction nommée `articlesJSON`.
- Nous allons importer les données du fichier **articles.json** avec l'objet JavaScript `XMLHttpRequest()` (<https://developer.mozilla.org/fr/docs/Web/API/XMLHttpRequest>), ce qui permet de récupérer des données via HTTP.
- Nous créons une variable nommée `httpRequestArticles` pour stocker la requête.
- La méthode `open()` permet de paramétrer cette requête.
- Le paramètre `GET` permet de spécifier le protocole HTTP utilisé.
- Le paramètre suivant est le chemin d'accès au fichier **.json** à importer.
- Le dernier paramètre, `true`, indique que le fichier doit être totalement chargé et récupéré avant d'en effectuer le traitement.
- La méthode `setRequestHeader()` permet de préciser le format des données importées.
- La propriété `onreadystatechange` permet d'indiquer ce que nous devons faire lorsque les données du serveur sont reçues. Dans cet exemple, nous créons une fonction anonyme.
- Ensuite, nous testons, `if()`, si la requête est terminée, `==4`, et si le statut est OK, `==200`.
- Si ces tests sont OK, nous créons une variable nommée `donneesJSON`.
- Dans cette nouvelle variable, nous stockons la réponse du serveur, `JSON.parse`, dans un objet. Dans cet objet, nous avons les données, sous forme de chaînes de caractères, `responseText`.
- Pour afficher cet objet, un simple `console.log(donneesJSON)` suffit.
- Enfin, nous indiquons qu'il n'y a pas d'envoi de données via la requête HTTP initiale, `httpRequestArticles.send(null)`.

Voici l’affichage obtenu dans la console, nous visualisons bien l’objet :



Pour afficher le contenu de cet objet, cliquez sur le triangle basculant devant **Object** :



Si nous souhaitons n’afficher qu’un seul élément, l’article 2 par exemple, il suffit de l’ajouter à l’objet JSON : `console.log(donneesJSON.article2);`.

Voici l’affichage obtenu :



Il en est de même pour n’afficher qu’une seule donnée, comme l’auteur par exemple : `console.log(donneesJSON.article2.auteur);`.

Voici l’affichage obtenu :



L’exemple à télécharger est dans le dossier **Chapitre-07-B/exemple-02.html**.

Installer le fichier `mustache.js`

Pour exploiter l’utilisation des templates avec Mustache, il faut dans un premier temps télécharger son script, afin de l’utiliser dans tous les exemples à venir.

Allez sur la page du site de **Mustache** : <http://mustache.github.io>



Logic-less templates.

Available in [Ruby](#), [JavaScript](#), [Python](#),
[Erlang](#), [Elixir](#), [PHP](#), [Perl](#), [Perl6](#), [Objective-C](#),
[Java](#), [C#/.NET](#), [Android](#), [C++](#), [CFEngine](#),
[Go](#), [Lua](#), [ooc](#), [ActionScript](#), [ColdFusion](#),
[Scala](#), [Clojure\[Script\]](#), [Fantom](#), [CoffeeScript](#),
[D](#), [Haskell](#), [XQuery](#), [ASP](#), [Io](#), [Dart](#), [Haxe](#),
[Delphi](#), [Racket](#), [Rust](#), [OCaml](#), [Swift](#), [Bash](#),
[Julia](#), [R](#), [Crystal](#), [Common Lisp](#), [Nim](#), [Pharo](#),
[Tcl](#), [C](#), [ABAP](#), and for [Elm](#)

Dans la liste des langages disponibles, cliquez sur le lien **JavaScript**.

Dans la liste des fichiers disponibles, téléchargez **mustache.js**.



Dans le dossier de votre site, enregistrez le fichier **mustache.js** à la racine de ce dossier ou dans un dossier nommé **javascript** ou **js**, comme vous le souhaitez.

Ensuite, dans chaque fichier HTML, nous devons faire le lien vers le fichier **mustache.js** :

```
<script src="mustache.js"></script>
```

Afficher des données internes simples

1. La structure de la page

Dans ce premier exemple, nous allons créer un template avec des données très simples. Nous allons avoir besoin de ces deux constituants principaux :

- des données internes simples au format **JSON**,
- un template au format **Mustache**.

Nous allons exécuter le rendu des données et les afficher dans un élément HTML de la page.

Voici le code de ce premier exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <div id="conteneur"></div>
    <script src="mustache.js"></script>
    <script>
      // Fournir les données internes en JSON
      var chienJSON = {"Nom": "Raja", "Couleur": "Brun", "Race":
"Labrador"} ;
      // Définir le template
```



```

        var leTemplate = "<h2>Nom du chien : {{Nom}}.</h2><p>Couleur du
chien : {{Couleur}}.</p><p>Race du chien : {{Race}}.</p>" ;
        // Exécuter le rendu
        var rendu = Mustache.render(leTemplate,chienJSON) ;
        // Afficher le rendu
        var leConteneur = document.getElementById('conteneur');
        leConteneur.innerHTML = rendu;
    </script>
</body>
</html>

```

2. Analyse de la structure

Analysons la structure de cette page.

Dans le <body>, nous avons une boîte <div> identifiée id="conteneur" qui est vide pour le moment. Cette boîte va servir de conteneur pour l’affichage du rendu des données.

```
<div id="conteneur"></div>
```

Ensuite, nous avons le lien vers le fichier **mustache.js** :

```
<script src="mustache.js"></script>
```

Puis, nous avons toute la partie du script qui va permettre d’afficher les données.

La première ligne permet d’avoir nos données JSON. Dans cet exemple, les données sont très simples, il n’y a que trois couples nom/valeur. Ces données sont stockées dans une variable nommée chienJSON :

```
var chienJSON = {"Nom":"Raja","Couleur":"Brun","Race":"Labrador"} ;
```

La deuxième ligne permet de définir le template. Ce template est défini dans une variable nommée leTemplate :

```
var leTemplate = "<h2>Nom du chien : {{Nom}}.</h2><p>Couleur du chien :
{{Couleur}}.</p><p>Race du chien : {{Race}}.</p>" ;
```

Vous voyez que nous indiquons toute la structure HTML nécessaire dans cette variable. Vous pourriez également ajouter des règles CSS si besoin était.

Vous notez également sûr la présence des balises Mustache qui font référence aux noms des données JSON. Les balises Mustache sont indiquées entre doubles accolades : {{Nom}}. Dans cet exemple, le contenu de cette balise va être remplacé par la valeur de l’élément Nom dans les données JSON. Il en est de même pour les balises {{Couleur}} et {{Race}}.

La troisième ligne permet d’effectuer le rendu des données JSON dans le template.

```
var rendu = Mustache.render(leTemplate,chienJSON) ;
```

Ce rendu est stocké dans une variable nommée rendu. Ce rendu est possible avec l’utilisation de la méthode render() de Mustache. Cette méthode utilise deux arguments :

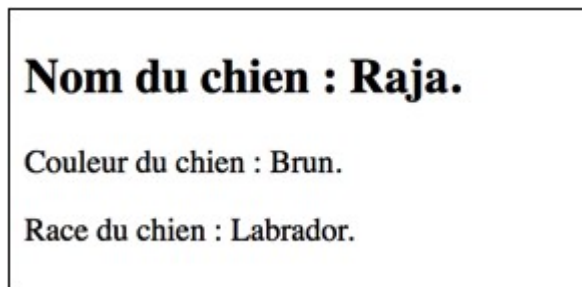
- Le premier indique le template à utiliser, la variable leTemplate dans cet exemple.
- Le deuxième indique les données à utiliser, la variable chienJSON dans cet exemple.

Enfin, la dernière ligne permet d’obtenir l’affichage de ce rendu dans le conteneur initial :

```
var leConteneur = document.getElementById('conteneur');
leConteneur.innerHTML = rendu;
```

Pour cela, nous créons une variable nommée `leConteneur` qui fait référence à la boîte `<div id="conteneur">`. Dans cet objet, nous insérons le code HTML, `innerHTML`, du rendu.

Voici l’affichage obtenu :



L'exemple à télécharger est dans le dossier **Chapitre-07-D/exemple-01.html**.

Afficher des données internes répétitives

1. La structure des données JSON

Dans ce deuxième exemple, nous allons utiliser des données répétitives. Nous reprenons l'exemple précédent, mais cette fois nous avons plusieurs chiens.

Voici les données indiquées au format JSON :

```
{"chiens": [
  {"Nom": "Raja", "Couleur": "Brun", "Race": "Labrador"},
  {"Nom": "Lord", "Couleur": "Beige", "Race": "Beagle"},
  {"Nom": "Yuky", "Couleur": "Noir", "Race": "Caniche"}
]
```

Le premier élément "chiens" est un tableau qui contient toutes ses données entre crochets [...]. Chaque donnée, chaque chien est défini entre accolades {...}. Pour chaque chien, nous retrouvons les couples nom/valeur.

2. La structure de la page

La structure de la page est très similaire à celle précédemment utilisée :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <div id="conteneur"></div>
    <script src="mustache.js"></script>
    <script>
      // Fournir les données internes en JSON
      var chiensJSON =
{"chiens": [{"Nom": "Raja", "Couleur": "Brun", "Race": "Labrador"}, {"Nom": "Lord",
```

```

"Couleur":"Beige","Race":"Beagle"}, {"Nom":"Yuky","Couleur":"Noir",
"Race":"Caniche"}] ] ;
    // Définir le template
    var leTemplate = "{{#chiens}}<h2>Nom du chien : {{Nom}}.</h2>
<p>Couleur du chien : {{Couleur}}.</p><p>Race du chien : {{Race}}.</p>
{{/chiens}}" ;
    // Exécuter le rendu
    var rendu = Mustache.render(leTemplate,chiensJSON) ;
    // Afficher le rendu
    var leConteneur = document.getElementById('conteneur');
    leConteneur.innerHTML = rendu;
</script>
</body>
</html>

```

Seule la définition du template change, puisqu'il faut afficher tous les chiens.

```

var leTemplate = "{{#chiens}}<h2>Nom du chien : {{Nom}}.</h2>
<p>Couleur du chien : {{Couleur}}.</p><p>Race du chien : {{Race}}.</p>
{{/chiens}}" ;

```

Pour afficher toutes les données répétitives, nous allons créer une "boucle" Mustache, une **section** pour reprendre la terminologie officielle. Pour ouvrir cette boucle, nous utilisons la balise `{{#chiens}}` qui reprend le nom du tableau dans les données JSON, préfixé par le caractère `#`. Pour fermer la boucle, c'est le caractère `/` qui est utilisé pour le préfixe : `{{/chiens}}`.

Mustache attend qu'il y ait une réponse à l'élément `{{#chiens}}`. Cette réponse peut être le booléen `true`, la valeur numérique `1` (équivalent au booléen `true`), une valeur renseignée ou une valeur provenant d'une réponse de données. C'est ce dernier cas qui est utilisé pour cet exemple, puisque dans les données JSON, l'élément `{{chiens}}` est bien renseigné par des valeurs.

Le reste du code est identique.

Voici l'affichage obtenu :

Nom du chien : Raja.

Couleur du chien : Brun.

Race du chien : Labrador.

Nom du chien : Lord.

Couleur du chien : Beige.

Race du chien : Beagle.

Nom du chien : Yuky.

Couleur du chien : Noir.

Race du chien : Caniche.

L'exemple à télécharger est dans le dossier **Chapitre-07-E/exemple-01.html**.

Afficher des données internes et un template séparé

Dans les deux exemples précédents, nous avons créé le template directement dans une variable. Cela peut être utilisé pour des projets ayant un design simple. Mais dès lors que le design est plus complexe, cette solution n'est pas très pratique.

Dans cet exemple, nous allons reprendre les données répétitives, nous allons "sortir" le template de la variable et le créer dans un élément `<script>` pour que cet élément ne soit pas affiché dans la fenêtre du navigateur, comme le serait un élément `<div>` par exemple.

Voici la structure de la page :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <div id="conteneur"></div>
    <script id="template" type="x-tmpl-mustache">
      {{#chiens}}
        <h2>Nom du chien : {{Nom}}.</h2>
        <p>Couleur du chien : {{Couleur}}.</p>
        <p>Race du chien : {{Race}}.</p>
      {{/chiens}}
    </script>
    <script src="mustache.js"></script>
    <script>
      // Fournir les données internes en JSON
      var chiensJSON =
{"chiens":[{"Nom":"Raja","Couleur":"Brun","Race":"Labrador"}, {"Nom":"Lord",
"Couleur":"Beige","Race":"Beagle"}, {"Nom":"Yuky","Couleur":"Noir",
"Race":"Caniche"}]} ;
      // Définir le template
      var leTemplate = document.getElementById('template').innerHTML ;
      // Exécuter le rendu
      var rendu = Mustache.render(leTemplate,chiensJSON) ;
      // Afficher le rendu
      var leConteneur = document.getElementById('conteneur') ;
      leConteneur.innerHTML = rendu;
    </script>
  </body>
</html>
```

Après la boîte `<div id="conteneur">` du conteneur, nous avons un nouvel élément `<script>` :

```
<script id="template" type="x-tmpl-mustache">
  {{#chiens}}
    <h2>Nom du chien : {{Nom}}.</h2>
    <p>Couleur du chien : {{Couleur}}.</p>
    <p>Race du chien : {{Race}}.</p>
  {{/chiens}}
</script>
```

Cet élément `<script>` possède un identifiant `id="template"` qui sera employé pour définir le template à utiliser. Il possède aussi un attribut `type` permettant de définir son type.

Dans cet élément `<script>`, nous retrouvons la structure du template précédent, mais avec une mise en forme nettement plus lisible et plus efficace à modifier. C'est un confort très appréciable.

Enfin, dernier point important, c'est la définition du template :

```
var leTemplate = document.getElementById('template').innerHTML ;
```

Avec cette structure du template séparé, il faut impérativement utiliser la propriété `innerHTML`, afin de ne récupérer que le contenu de l'élément HTML identifié `template` :

```
{{#chiens}}  
  <h2>Nom du chien : {{Nom}}.</h2>  
  <p>Couleur du chien : {{Couleur}}.</p>  
  <p>Race du chien : {{Race}}.</p>  
{{/chiens}}
```

C'est uniquement les balises Mustache et HTML qui seront utilisées dans le rendu.

Si vous n'indiquez pas la propriété `innerHTML`, vous allez récupérer tout l'élément HTML :

```
<script id="template" type="x-tmpl-mustache">  
  {{#chiens}}  
    <h2>Nom du chien : {{Nom}}.</h2>  
    <p>Couleur du chien : {{Couleur}}.</p>  
    <p>Race du chien : {{Race}}.</p>  
  {{/chiens}}  
</script>
```

Et le rendu ne fonctionnera pas.

Bien sûr, l'affichage obtenu est strictement identique à l'exemple précédent.

L'exemple à télécharger est dans le dossier **Chapitre-07-F/exemple-01.html**.

Décomposer le template en blocs

1. Les objectifs

Dans l'exemple précédent, nous avons placé le template dans un élément `<script>` séparé, afin de mieux séparer les composants de la structure de la page. Mais nous pouvons aller plus loin encore, en décomposant le template en blocs fonctionnels. Dans la terminologie de Mustache, les blocs de template séparés du template principal s'appellent des **partials**.

Il se peut que votre template soit constitué de nombreuses parties fonctionnelles distinctes. Dans ce cas, il peut vite devenir confus et verbeux. Il est alors préférable de le décomposer en plusieurs blocs séparés, chaque bloc s'occupant d'une fonctionnalité précise.

Dans notre exemple, nous allons séparer la partie affichant la race des chiens dans un bloc distinct. Nous aurons ainsi deux templates, le premier gérant l'affichage du nom et de la couleur du chien, le deuxième affichant la race du chien. Bien sûr, cet exemple reste simple, à vous d'adapter cette démarche à vos projets plus complexes.

2. Les templates

Dans le template initial, nous voulons sortir la partie gérant la race des chiens.

Voici le template initial :

```
<script id="template" type="x-tmpl-mustache">
  {{#chiens}}
    <h2>Nom du chien : {{Nom}}.</h2>
    <p>Couleur du chien : {{Couleur}}.</p>
    <p>Race du chien : {{Race}}.</p>
  {{/chiens}}
</script>
```

Nous souhaitons donc retirer la ligne `<p>Race du chien : {{Race}}.</p>`.

Nous créons un deuxième template, juste sous le premier :

```
<script id="race" type="x-tmpl-mustache">
  <p><em>Race du chien</em> : <strong>{{Race}}</strong>.</p>
</script>
```

Ce template est dans un élément HTML `<script>` ayant bien sûr un identifiant unique : `id="race"`.

Dans ce template, nous utilisons les éléments HTML et les balises Mustache nécessaires à la mise en forme voulue.

Maintenant, nous allons dans le premier template faire référence au deuxième template, au deuxième bloc :

```
<script id="template" type="x-tmpl-mustache">
  {{#chiens}}
    <h2>Nom du chien : {{Nom}}.</h2>
    <p>Couleur du chien : {{Couleur}}.</p>
    {{> race}}
  {{/chiens}}
</script>
```

La référence à un autre bloc se fait avec une balise Mustache spécifique : `{{> race}}`. Dans la balise, entre les doubles accolades, nous avons le caractère `>` suivi par un espace. Puis, nous devons indiquer l'identifiant unique du template, du bloc à insérer, `race` dans cet exemple.

Nous devons maintenant définir ce deuxième template, juste après le premier :

```
var laRace = document.getElementById('race').innerHTML ;
```

La syntaxe est la même que dans l'exemple précédent. La variable `laRace` fait bien référence à l'élément HTML ayant l'identifiant `race`.

3. Le rendu

Le rendu par Mustache doit être maintenant modifié, afin de prendre en compte le nouveau template.

```
var rendu = Mustache.render(leTemplate,chiensJSON, {race: laRace}) ;
```

Dans la variable de rendu initiale rendu, nous ajoutons un troisième argument à la méthode render. Nous indiquons entre simples accolades {...} l'identifiant du template HTML race, suivi par le caractère :, un espace et le nom de la variable définissant ce deuxième template laRace.

Voici l'affichage obtenu :



4. Le code complet de l'exemple

Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <div id="conteneur"></div>
    <script id="template" type="x-tmpl-mustache">
      {{#chiens}}
        <h2>Nom du chien : {{Nom}}.</h2>
        <p>Couleur du chien : {{Couleur}}.</p>
        {{> race}}
      {{/chiens}}
    </script>
    <script id="race" type="x-tmpl-mustache">
      <p><em>Race du chien</em> : <strong>{{Race}}</strong>.</p>
    </script>
    <script src="mustache.js"></script>
    <script>
      // Fournir les données internes en JSON
      var chiensJSON =
{"chiens":[{"Nom":"Raja","Couleur":"Brun","Race":"Labrador"}, {"Nom":"Lord",
"Couleur":"Beige","Race":"Beagle"}, {"Nom":"Yuky","Couleur":"Noir",
"Race":"Caniche"}]} ;
      //console.log(chiensJSON);
      // Définir les templates
```

```

        var leTemplate = document.getElementById('template').innerHTML ;
        var laRace = document.getElementById('race').innerHTML ;
        // Exécuter le rendu
        var rendu = Mustache.render(leTemplate,chiensJSON, {race: laRace})
        // Afficher le rendu
        var leConteneur = document.getElementById('conteneur') ;
        leConteneur.innerHTML = rendu;

</script>
</body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-07-G/exemple-01.html**.

Afficher des données extérieures et utiliser un template séparé

1. Les objectifs

Dans ce dernier exemple, nous allons concevoir un template séparé dans un élément HTML `<script>` qui utilisera des données extérieures au format JSON.



Attention, dans cet exemple, nous allons à nouveau importer des données extérieures au fichier .html avec le protocole HTTP, vous devez donc utiliser un serveur web local, de type MAMP, WAMP, XAMPP, selon votre convenance.

2. Les données extérieures

Toutes les données de cet exemple sont dans un fichier **articles.json** :

```

{
  "articles":[
    {
      "titre": "Lorem Ipsum",
      "contenu": "Aenean lacinia bibendum nulla sed...",
      "date": "08/06/2018",
      "auteur": "Christophe"
    },
    {
      "titre": "Cursus Malesuada Amet",
      "contenu": "Praesent commodo cursus magna...",
      "date": "11/06/2018",
      "auteur": "Christine"
    },
    {
      "titre": "Maecenas faucibus",
      "contenu": "Vestibulum id ligula porta felis...",
      "date": "18/06/2018",
      "auteur": "Mathis"
    }
  ]
}

```


L'élément premier est la donnée nommée "articles". C'est un type tableau qui contient ses données entre les deux crochets [...]. Chaque entrée du tableau est placée entre accolades {...} et possède les mêmes couples nom/valeur : "titre", "contenu", "date" et "auteur".

3. La structure de la page

Nous avons trois parties principales dans la page HTML : le template, l'importation des données et le traitement de ces données pour leur affichage dans le template.

Voici le code complet de la page HTML :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
    <style>
      .contenu {
        border-left: solid 5px #eee;
        padding-left: 10px;
      }
      .metadonnees {
        font-size: small;
        font-style: italic;
      }
    </style>
  </head>
  <body>
    <h1>Les articles</h1>
    <div id="conteneur"></div>
    <script id="template" type="x-tmpl-mustache">
      {{#articles}}
        <h2 class="titre">{{titre}}</h2>
        <p class="contenu">
          {{contenu}}
          <br>
          <span class="metadonnees">Rédigé le {{date}} par
{{auteur}}</span>
        </p>
      {{/articles}}
    </script>
    <script src="mustache.js"></script>
    <script>
      // Récupérer les données JSON extérieures
      var lesArticles ;
      function donneesJSON() {
        var httpRequestArticles = new XMLHttpRequest();
        httpRequestArticles.open("GET", "articles.json", true);
        httpRequestArticles.setRequestHeader("Content-type",
"application/json");
        httpRequestArticles.onreadystatechange = function() {
          if (httpRequestArticles.readyState == 4 &&
httpRequestArticles.status == 200) {
            lesArticles =
JSON.parse(httpRequestArticles.responseText);
            afficheArticles();
          }
        }
        httpRequestArticles.send(null);
      }
      donneesJSON();
    </script>
  </body>
</html>
```

```

        // Fonction d'affichage des données
        function afficheArticles() {
            // Définir le template
            var leTemplate =
document.getElementById('template').innerHTML ;
            // Exécuter le rendu
            var rendu = Mustache.render(leTemplate, lesArticles) ;
            // Afficher le rendu
            var leConteneur = document.getElementById('conteneur') ;
            leConteneur.innerHTML = rendu;
        }
    </script>
</body>
</html>

```

4. Le template

Le template est placé dans un élément `<script>` ayant l'identifiant `id="template"` :

```

<script id="template" type="x-tmpl-mustache">
    {{#articles}}
        <h2 class="titre">{{titre}}</h2>
        <p class="contenu">
            {{contenu}}
            <br>
            <span class="metadonnees">Rédigé le {{date}} par
{{auteur}}</span>
        </p>
    {{/articles}}
</script>

```

Nous créons une boucle avec la balise Mustache `{{#articles}}` qui est le nom du tableau dans les données JSON. Donc cet élément est bien renseigné.

Ensuite, nous utilisons les éléments HTML et CSS voulus, avec les balises Mustache pour créer le template.

Voici les règles CSS utilisées dans ce template :

```

<style>
    .contenu {
        border-left: solid 5px #eee;
        padding-left: 10px;
    }
    .metadonnees {
        font-size: small;
        font-style: italic;
    }
</style>

```

5. L'importation des données JSON

Nous reprenons la même structure que celle vue en début de chapitre :

```

var lesArticles ;
function donneesJSON() {
    var httpRequestArticles = new XMLHttpRequest();
    httpRequestArticles.open("GET", "articles.json", true);
    httpRequestArticles.setRequestHeader("Content-type", "application/json");
    httpRequestArticles.onreadystatechange = function() {

```

```

    if (httpRequestArticles.readyState == 4 && httpRequestArticles.status
== 200) {
        lesArticles = JSON.parse(httpRequestArticles.responseText);
        afficheArticles();
    }
    httpRequestArticles.send(null);
}
donneesJSON();

```

Voyons les particularités de ce script.

- À la première ligne, nous définissons une variable globale, `lesArticles`, sans valeur initiale, puisqu'elle est créée en dehors de toute fonction. Elle sera ainsi utilisable dans tout le script.
- Comme précédemment, nous créons une fonction nommée `donneesJSON()` pour importer les données stockées dans le fichier **articles.json**.
- Dans la fonction `if()` qui teste les réponses du serveur, nous utilisons la variable globale `lesArticles` pour stocker les données textuelles JSON extérieures récupérées.
- Puis le test `if()` se termine par l'appel à une fonction nommée `afficheArticles()`.

6. Le traitement des données

La dernière partie du script s'occupe du traitement des données avec la fonction `afficheArticles()` :

```

function afficheArticles() {
    // Définir le template
    var leTemplate = document.getElementById('template').innerHTML ;
    // Exécuter le rendu
    var rendu = Mustache.render(leTemplate, lesArticles) ;
    // Afficher le rendu
    var leConteneur = document.getElementById('conteneur') ;
    leConteneur.innerHTML = rendu;
}

```

Cette fonction est appelée depuis la fonction d'importation des données, dans le test `if()` des réponses du serveur.

Comme nous l'avons déjà vu précédemment, nous définissons plusieurs variables :

- la variable `leTemplate` pour référencer le template,
- la variable `rendu` pour traiter les données importées par Mustache,
- la variable `leConteneur` pour afficher le rendu des données dans le template.

Voici l'affichage obtenu :

Les articles

Lorem Ipsum

Aenean lacinia bibendum nulla sed consectetur. Maecenas faucibus mollis interdum. Sed posuere consectetur est at lobortis. Nullam quis risus eget urna mollis ornare vel eu leo, Integer posuere erat a ante venenatis dapibus posuere velit aliquet. Donec id elit non mi porta gravida at eget metus.

Rédigé le 08/06/2018 par Christophe

Cursus Malesuada Amet

Prasent commodo cursus magna, vel scelerisque nisl consectetur et. Donec sed odio dui. Maecenas sed diam eget risus varius blandit sit amet non magna. Nullam id dolor id nibh ultricies vehicula ut id elit. Curabitur blandit tempus porttitor. Morbi leo risus, porta ac consectetur ac, vestibulum at eros. Integer posuere erat a ante venenatis dapibus posuere velit aliquet. Donec id elit non mi porta gravida at eget metus.

Rédigé le 11/06/2018 par Christine

Maecenas faucibus

Vestibulum id ligula porta felis euismod semper. Aenean lacinia bibendum nulla sed consectetur. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Morbi leo risus, porta ac consectetur ac, vestibulum at eros. Nulla vitae elit libero, a pharetra augue. Duis mollis, est non commodo luctus, nisi erat porttitor ligula, eget lacinia odio sem nec elit. Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum.

Rédigé le 18/06/2018 par Mathis

L'exemple à télécharger est dans le dossier **Chapitre-07-H/exemple-01.html**.

Présentation du moteur de rendu Handlebars

Handlebars est un moteur de rendu, tout comme Mustache vu dans le chapitre précédent, dont il est assez proche. D'ailleurs, Handlebars est globalement compatible avec les templates de Mustache.

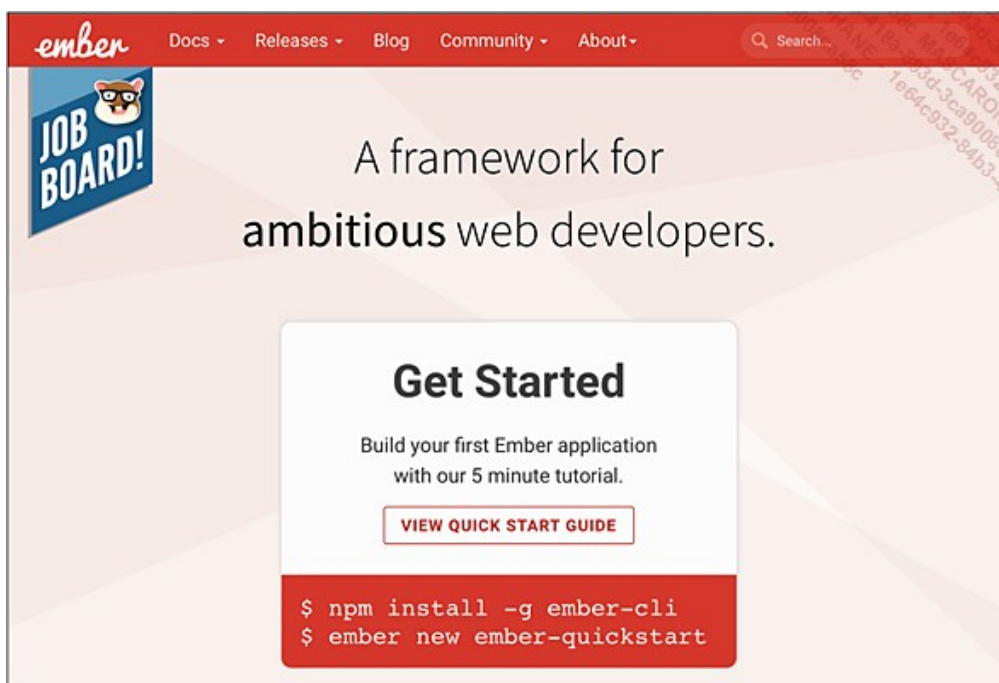
Voici l'URL de son site : <http://handlebarsjs.com>



Une des grandes différences avec Mustache est que Handlebars nous propose des fonctions de logique (des helpers dans la terminologie anglaise de Handlebars) pour concevoir nos templates, avec des instructions each, if et with qui vont nous offrir des fonctionnalités supplémentaires appréciables.

Nous n'allons pas revenir sur les sources de données, le principe est strictement identique à Mustache, vu dans le chapitre précédent.

Notez que Handlebars fait partie du framework **Ember**, qui permet le développement d'applications JavaScript très structurées et performantes.

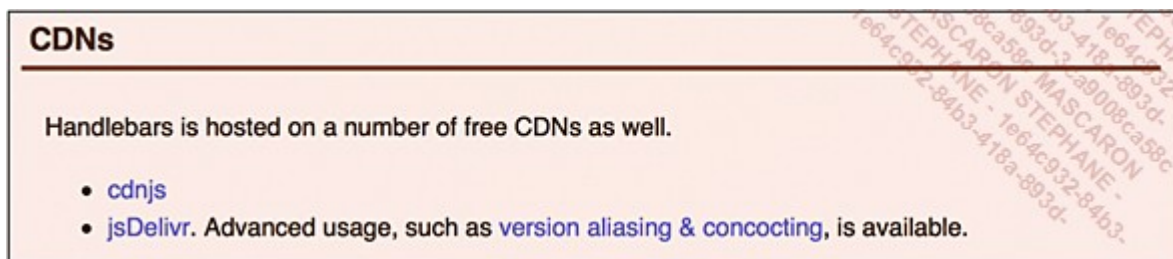


Installer Handlebars

La première étape consiste bien sûr à installer Handlebars. La meilleure solution pour exploiter Handlebars dans vos projets web affichés dans notre navigateur est de l'installer via un CDN.

Sur la page d'accueil du site Handlebars, cliquez sur le bouton **Installation**.

Sur cette page, tout en bas, dans la zone **CDNs**, cliquez sur le lien **cdnjs**.



Dans le site de cdnjs, dans la page de Handlebars, sélectionnez le lien <https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.11/handlebars.min.js>.



Dans le fichier **.html** voulu, avant la balise de fermeture `</body>`, insérez un élément `<script>` et renseignez l'attribut `src` avec cette URL :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    ...
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.11/
handlebars.min.js"></script>
  </body>
</html>
```

Concevoir un template avec des données répétitives

1. Les données répétitives

Pour ce premier exemple, nous allons utiliser des données répétitives au format JSON, en reprenant l'exemple de la liste des chiens du chapitre précédent. Ces données sont placées en interne, dans un élément `<script>` :

```
<script>
  var chiensJSON = {"chiens":[
    {"Nom":"Raja","Couleur":"Brun","Race":"Labrador"},
    {"Nom":"Lord","Couleur":"Beige","Race":"Beagle"},
    {"Nom":"Yuky","Couleur":"Noir","Race":"Caniche"}
  ]} ;
```

</script>

2. Le template

Pour concevoir nos templates, nous allons à nouveau utiliser l'élément HTML `<script>`, avec un identifiant et un type spécifique à Handlebars.

```
<script id="template" type="text/x-handlebars-template">
```

Comme précédemment pour créer les templates, c'est une syntaxe à double accolades qu'il faut utiliser.

Pour effectuer des boucles, nous allons utiliser une syntaxe propre à Handlebars. Dans notre exemple de données, c'est l'élément `chiens` qui doit être répété autant de fois qu'il y a de chiens. Nous devons utiliser le mot-clé `each` dans les balises.

Voici la syntaxe pour la balise d'ouverture : `{{#each chiens}}`. Nous devons utiliser comme préfixe à `each` le caractère `#`, suivi d'un espace et du nom de l'élément à répéter, `chiens`.

Voici la syntaxe pour la balise de fermeture : `{{/each}}`. Nous utilisons comme préfixe à `each` le caractère `/`. Il ne faut pas indiquer le nom de l'élément à répéter.

Voilà le code du template :

```
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    <h2>Nom du chien : {{Nom}}.</h2>
    <p>Couleur du chien : {{Couleur}}.</p>
    <p>Race du chien : {{Race}}.</p>
  {{/each}}
</script>
```

3. Le rendu des données dans le template

Pour effectuer le rendu, nous devons indiquer à Handlebars quel est le template à utiliser, dans un élément `<script>` :

```
var leTemplate = document.getElementById('template').innerHTML ;
```

Ensuite, nous devons compiler ce template, `leTemplate`, avec la méthode `compile()` de Handlebars :

```
var compileTemplate = Handlebars.compile(leTemplate) ;
```

Puis, nous devons exécuter le rendu des données `chiensJSON` dans la compilation du template `compileTemplate` :

```
var rendu = compileTemplate(chiensJSON);
```

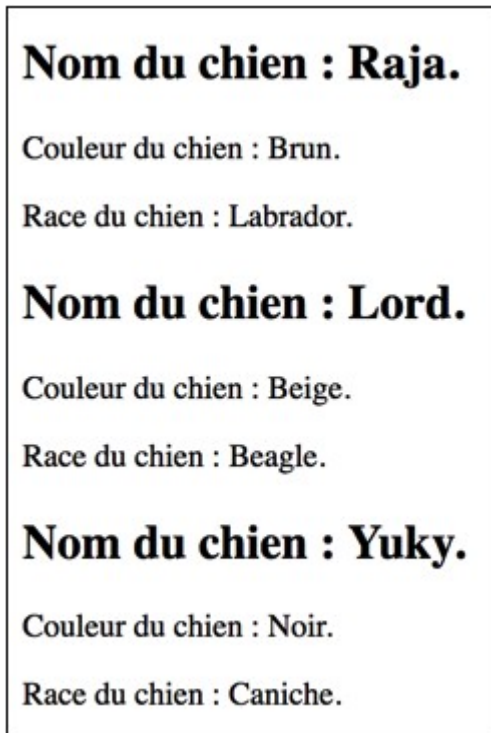
Maintenant nous pouvons nous occuper de l'affichage, en définissant le conteneur qui va afficher le rendu :

```
var leConteneur = document.getElementById('conteneur') ;
```

La dernière étape consiste à afficher dans le conteneur, `leConteneur`, le rendu des données, `rendu` :

```
leConteneur.innerHTML = rendu ;
```

Voici l’affichage obtenu :



4. Le code complet de cet exemple

Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <div id="conteneur"></div>
    <script id="template" type="text/x-handlebars-template">
      {{#each chiens}}
        <h2>Nom du chien : {{Nom}}.</h2>
        <p>Couleur du chien : {{Couleur}}.</p>
        <p>Race du chien : {{Race}}.</p>
      {{/each}}
    </script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/4.0.11/
handlebars.min.js"></script>
    <script>
      // Définir les données internes en JSON
      var chiensJSON = {"chiens":[{"Nom":"Raja",
"Couleur":"Brun","Race":"Labrador"}, {"Nom":"Lord",
"Couleur":"Beige","Race":"Beagle"}, {"Nom":"Yuky",
"Couleur":"Noir","Race":"Caniche"}]};
      // Définir et compiler le template
      var leTemplate = document.getElementById('template').innerHTML ;
      var compileTemplate = Handlebars.compile(leTemplate) ;
      // Exécuter le rendu
      var rendu = compileTemplate(chiensJSON);
      // Définir le conteneur
      var leConteneur = document.getElementById('conteneur') ;
      // Afficher le rendu
```



```

        leConteneur.innerHTML = rendu ;
    </script>
</body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-08-C/exemple-01.html**.

Décomposer des templates

1. Concevoir les templates

Comme pour Mustache, avec Handlebars nous allons pouvoir décomposer des templates complexes en plusieurs blocs fonctionnels, des partials dans la terminologie anglaise de Handlebars. Mais Handlebars nous offre plus de facilités que Mustache.

Nous allons reprendre l'exemple précédent en retirant du template principal l'affichage concernant la race du chien, pour le placer dans un bloc séparé.

Voici le template principal initial :

```

<script id="template" type="text/x-handlebars-template">
    {{#each chiens}}
        <h2>Nom du chien : {{Nom}}.</h2>
        <p>Couleur du chien : {{Couleur}}.</p>
        <p>Race du chien : {{Race}}.</p>
    {{/each}}
</script>

```

C'est donc la ligne `<p>Race du chien : {{Race}}.</p>` que nous allons retirer.

Nous créons un deuxième template pour ce bloc séparé :

```

<script id="race" type="text/x-handlebars-template">
    <p><em>Race du chien</em> : <strong>{{Race}}</strong>.</p>
</script>

```

Cet élément `<script>` possède son identifiant unique `id="race"`. Son contenu reprend la syntaxe du template initial, avec une mise en forme HTML supplémentaire.

Il nous faut maintenant faire appel à ce bloc dans le template principal :

```

<script id="template" type="text/x-handlebars-template">
    {{#each chiens}}
        <h2>Nom du chien : {{Nom}}.</h2>
        <p>Couleur du chien : {{Couleur}}.</p>
        {{> raceChien}}
    {{/each}}
</script>

```

La ligne d'affichage de la race doit faire appel au bloc séparé avec une balise nommée. Cette balise nommée contient le caractère `>` suivi par un espace. Ensuite nous avons le nom de référence de cette balise : `raceChien`. Voici la syntaxe complète : `{{> raceChien}}`.

2. Définir le nouveau template

Maintenant, dans le script principal, nous allons définir ce nouveau template et définir ce nouveau bloc, ce nouveau partial.

```
<script>
    var laRace = document.getElementById('race').innerHTML ;
    Handlebars.registerPartial('raceChien', laRace) ;
    ...
</script>
```

La première ligne permet de définir le nouveau template dans une variable nommée laRace. Cette variable fait référence à l'élément dont l'identifiant est race.

La deuxième ligne permet d'enregistrer le nouveau bloc avec la méthode registerPartial() de Handlebars. Le premier argument est le nom de la balise de référence dans le template principal, raceChien, défini précédemment. Le deuxième argument est le nom du template du nouveau bloc, laRace, défini à la ligne précédente.

Et il n'y a rien de plus à faire. C'est une syntaxe très concise et facile à utiliser. Bien sûr, si la page conçue le permet, vous pourrez utiliser ce bloc séparé à d'autres endroits dans la page.

Voici l'affichage obtenu :



3. Le code complet

Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Ma page</title>
    </head>
    <body>
```

```

<div id="conteneur"></div>
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    <h2>Nom du chien : {{Nom}}.</h2>
    <p>Couleur du chien : {{Couleur}}.</p>
    {{> raceChien}}
  {{/each}}
</script>
<script id="race" type="text/x-handlebars-template">
  <p><em>Race du chien</em> : <strong>{{Race}}</strong>.</p>
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/
4.0.11/handlebars.min.js"></script>
<script>
  // Définir le bloc externe
  var laRace = document.getElementById('race').innerHTML ;
  Handlebars.registerPartial('raceChien', laRace) ;
  // Définir les données internes en JSON
  var chiensJSON =
{"chiens":[{"Nom":"Raja", "Couleur":"Brun", "Race":"Labrador"},
{"Nom":"Lord", "Couleur":"Beige", "Race":"Beagle"},
{"Nom":"Yuky", "Couleur":"Noir", "Race":"Caniche"}]} ;
  // Définir et compiler le template
  var leTemplate = document.getElementById('template').innerHTML ;
  var compileTemplate = Handlebars.compile(leTemplate) ;
  // Exécuter le rendu
  var rendu = compileTemplate(chiensJSON);
  // Définir le conteneur
  var leConteneur = document.getElementById('conteneur') ;
  // Afficher le rendu
  leConteneur.innerHTML = rendu ;
</script>
</body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-08-D/exemple-01.html**.

Les helpers

Handlebars nous propose des facilités pour utiliser du code qui se répète de nombreuses fois dans la conception des templates. Ces facilités, ces aides, se nomment des helpers dans la terminologie anglaise de Handlebars.

Nous allons mettre en place deux exemples. Un premier helper permettra d'appliquer une mise en forme répétitive et un deuxième permettra une concaténation.

Utiliser un helper pour une mise en forme répétitive

1. La répétition des données

Nous allons reprendre l'exemple précédent en modifiant quelque peu sa structure afin de pouvoir mettre en place le premier helper.

Voici le template principal :

```
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    <h2>Nom du chien : {{Nom}}.</h2>
    <p>Couleur du chien : {{Couleur}}.</p>
    {{> raceChien}}
  {{/each}}
</script>
```

Nous souhaitons mettre en gras les valeurs de chaque élément du tableau : le Nom (dans un élément `<p>` et non plus dans un élément `<h2>` comme cela était le cas initialement) et la Couleur. Pour mettre en place cela, nous pourrions insérer un élément HTML `` pour chacune de ces balises Handlebars :

```
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    <p>Nom du chien : <strong>{{Nom}}</strong>.</p>
    <p>Couleur du chien : <strong>{{Couleur}}</strong>.</p>
    {{> raceChien}}
  {{/each}}
</script>
```

Vous voyez que cela est très répétitif. Et si la mise en forme est plus complexe, le code deviendrait très verbeux.

Il en est de même pour le bloc initial :

```
<script id="race" type="text/x-handlebars-template">
  <p><em>Race du chien</em> : <strong>{{Race}}</strong>.</p>
</script>
```

2. Définir le helper

Nous allons donc définir un helper qui va permettre une mise en gras des balises voulues dans les templates.

C'est dans le script principal que nous allons définir ce helper :

```
<script>
  Handlebars.registerHelper('gras', function(texte){
    return '<strong>'+texte+'</strong>' ;
  })
  ...
</script>
```

Analysons ce code :

- Nous utilisons la méthode `registerHelper()` de Handlebars.
- Le premier argument est le nom de ce helper, `gras` dans cet exemple. Ce nom sera ensuite utilisé dans les balises qu'il faut mettre en gras.
- Le deuxième argument est une fonction anonyme, `function()`. Cette fonction utilise un argument nommé `texte`. Cet argument sera remplacé par le texte qu'il faut mettre en gras.
- La fonction renvoie `return`, la balise d'ouverture ``, concaténée avec le texte et concaténée avec la balise de fermeture ``.

3. Appliquer le helper

Maintenant que le helper gras est défini, dans les templates, nous allons l'appliquer à toutes les balises Handlebars voulues.

Dans le template principal :

```
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    <p>Nom du chien : {{gras Nom}}.</p>
    <p>Couleur du chien : {{gras Couleur}}.</p>
    {{> raceChien}}
  {{/each}}
</script>
```

Et dans le bloc séparé :

```
<script id="race" type="text/x-handlebars-template">
  <p><em>Race du chien</em> : {{gras Race}}.</p>
</script>
```

La syntaxe est la suivante : dans la balise concernée, nous saisissons le nom du helper, gras, suivi d'un espace et du nom de l'élément de données à afficher.

Voici l'affichage obtenu :

Nom du chien : Raja.

Couleur du chien : Brun.

Race du chien : Labrador.

Nom du chien : Lord.

Couleur du chien : Beige.

Race du chien : Beagle.

Nom du chien : Yuky.

Couleur du chien : Noir.

Race du chien : Caniche.

Cet affichage est tout à fait normal, car **Handlebars** affiche tout ce qu'il y a dans le helper. Et comme nous avons indiqué des balises HTML, elles sont affichées.

Nous devons donc masquer ces balises HTML, en effectuant un "échappement".

Cet échappement est une fonction classique du JavaScript et donc de Handlebars (et de Mustache aussi).

Pour ce faire, il faut ajouter une paire d'accolades aux balises concernées : {{{gras Nom}}}.

Voici le template principal corrigé :

```
<script id="template" type="text/x-handlebars-template">
  {{{#each chiens}}}
```

```

        <p>Nom du chien : {{{gras Nom}}}.</p>
        <p>Couleur du chien : {{{gras Couleur}}}.</p>
        {{> raceChien}}
    {{/each}}
</script>

```

Et dans le bloc séparé :

```

<script id="race" type="text/x-handlebars-template">
    <p><em>Race du chien</em> : {{{gras Race}}}.</p>
</script>

```

Voici l’affichage obtenu :

Nom du chien : **Raja.**

Couleur du chien : **Brun.**

Race du chien : **Labrador.**

Nom du chien : **Lord.**

Couleur du chien : **Beige.**

Race du chien : **Beagle.**

Nom du chien : **Yuky.**

Couleur du chien : **Noir.**

Race du chien : **Caniche.**

4. Le code complet

Voici le code complet de cet exemple :

```

<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>Ma page</title>
    </head>
    <body>
        <div id="conteneur"></div>
        <script id="template" type="text/x-handlebars-template">
            {{#each chiens}}
                <p>Nom du chien : {{{gras Nom}}}.</p>
                <p>Couleur du chien : {{{gras Couleur}}}.</p>
                {{> raceChien}}
            {{/each}}
        </script>
        <script id="race" type="text/x-handlebars-template">
            <p><em>Race du chien</em> : {{{gras Race}}}.</p>
        </script>
        <script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/
4.0.11/handlebars.min.js"></script>
        <script>
            // Définir le helper pour mettre en gras
            Handlebars.registerHelper('gras',function(texte){

```

```

        return '<strong>'+texte+'</strong>' ;
    })
    // Définir le bloc externe
    var laRace = document.getElementById('race').innerHTML ;
    Handlebars.registerPartial('raceChien', laRace) ;
    // Définir les données internes en JSON
    var chiensJSON =
{"chiens":[{"Nom":"Raja","Couleur":"Brun","Race":"Labrador"},
{"Nom":"Lord","Couleur":"Beige","Race":"Beagle"},
{"Nom":"Yuky","Couleur":"Noir","Race":"Caniche"}]} ;
    // Définir et compiler le template
    var leTemplate = document.getElementById('template').innerHTML ;
    var compileTemplate = Handlebars.compile(leTemplate) ;
    // Exécuter le rendu
    var rendu = compileTemplate(chiensJSON);
    // Définir le conteneur
    var leConteneur = document.getElementById('conteneur') ;
    // Afficher le rendu
    leConteneur.innerHTML = rendu ;
</script>
</body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-08-F/exemple-01.html**.

Utiliser un helper pour une concaténation

1. La concaténation des données

Dans ce deuxième exemple de helper, nous voulons concaténer deux entrées des données JSON. Dans ces données, nous ajoutons une nouvelle entrée nommée Proprietaire :

```

var chiensJSON = {"chiens":[
    {"Nom":"Raja","Couleur":"Brun","Race":"Labrador",
      "Proprietaire": {"NomPro":"Durand","PrenomPro":"Paul"}
    },
    {"Nom":"Lord","Couleur":"Beige","Race":"Beagle",
      "Proprietaire": {"NomPro":"Leconte","PrenomPro":"Valérie"}
    },
    {"Nom":"Yuky","Couleur":"Noir","Race":"Caniche",
      "Proprietaire": {"NomPro":"Tralent","PrenomPro":"Julie"}
    }
]} ;

```

Cette nouvelle entrée est composée de deux couples nom/valeur, NomPro et PrenomPro.

2. Définir le helper

C'est bien sûr dans le script principal que nous allons définir ce helper :

```

Handlebars.registerHelper('nomComplet',function(proprio){
    return proprio.PrenomPro + ' ' + proprio.NomPro ;
})

```

Le helper est nommé nomComplet et il concatène les valeurs de PrenomPro et NomPro.

3. Appliquer le helper

Maintenant que le helper `nomComplet` est défini, nous allons l'ajouter dans une nouvelle balise du template principal :

```
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    <p>Nom du chien : {{{gras Nom}}}.</p>
    <p>Couleur du chien : {{{gras Couleur}}}.</p>
    {{> raceChien}}
    <p>Le propriétaire : {{nomComplet Proprietaire}}.</p>
  {{/each}}
</script>
```

Après la ligne affichant la race du chien, nous ajoutons la balise `{{Proprietaire}}` de la nouvelle entrée des données. Nous ajoutons devant ce nom le nom du helper précédemment créé `nomComplet`, pour obtenir cette balise complète : `{{nomComplet Proprietaire}}`.

Voici l'affichage obtenu :

Nom du chien : **Raja**.

Couleur du chien : **Brun**.

Race du chien : **Labrador**.

Le propriétaire : Paul Durand.

Nom du chien : **Lord**.

Couleur du chien : **Beige**.

Race du chien : **Beagle**.

Le propriétaire : Valérie Leconte.

Nom du chien : **Yuky**.

Couleur du chien : **Noir**.

Race du chien : **Caniche**.

Le propriétaire : Julie Tralent.

4. Le code complet

Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <div id="conteneur"></div>
    <script id="template" type="text/x-handlebars-template">
      {{#each chiens}}
```



```

        <p>Nom du chien : {{{gras Nom}}}.</p>
        <p>Couleur du chien : {{{gras Couleur}}}.</p>
        {{> raceChien}}
        <p>Le propriétaire : {{nomComplet Proprietaire}}.</p>
    {{/each}}
</script>
<script id="race" type="text/x-handlebars-template">
    <p><em>Race du chien</em> : {{{gras Race}}}.</p>
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/
4.0.11/handlebars.min.js"></script>
<script>
    // Définir le helper pour mettre en gras
    Handlebars.registerHelper('gras',function(texte){
        return '<strong>'+texte+'</strong>' ;
    })
    // Définir le helper pour le nom complet du propriétaire
    Handlebars.registerHelper('nomComplet',function(proprio){
        return proprio.PrenomPro + ' ' + proprio.NomPro ;
    })
    // Définir le bloc externe
    var laRace = document.getElementById('race').innerHTML ;
    Handlebars.registerPartial('raceChien',laRace) ;
    // Définir les données internes en JSON
    var chiensJSON = {"chiens":[
        {"Nom":"Raja","Couleur":"Brun","Race":"Labrador",
        "Proprietaire": {"NomPro":"Durand","PrenomPro":"Paul"}
        },
        {"Nom":"Lord","Couleur":"Beige","Race":"Beagle",
        "Proprietaire": {"NomPro":"Leconte","PrenomPro":"Valérie"}
        },
        {"Nom":"Yuky","Couleur":"Noir","Race":"Caniche",
        "Proprietaire": {"NomPro":"Tralent","PrenomPro":"Julie"}
        }
    ]} ;
    // Définir et compiler le template
    var leTemplate = document.getElementById('template').innerHTML ;
    var compileTemplate = Handlebars.compile(leTemplate) ;
    // Exécuter le rendu
    var rendu = compileTemplate(chiensJSON);
    // Définir le conteneur
    var leConteneur = document.getElementById('conteneur') ;
    // Afficher le rendu
    leConteneur.innerHTML = rendu ;
</script>
</body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-08-G/exemple-01.html**.

Utiliser un helper conditionnel

1. Utiliser une condition de test

Handlebars nous permet de faire des tests conditionnels avec le mot-clé if. Des données ne seront pas affichées, si une donnée spécifiée n'est pas définie, si le test if sur la donnée testée renvoie false, undefined, null, "", 0, ou [].

Dans cet exemple, nous allons modifier nos données sources en supprimant l'entrée Nom pour le deuxième chien :

```
var chiensJSON = {"chiens":[
  {"Nom":"Raja","Couleur":"Brun","Race":"Labrador"},
  {"Couleur":"Beige","Race":"Beagle"},
  {"Nom":"Yuky","Couleur":"Noir","Race":"Caniche"}
]}
```

Voici l’affichage obtenu sans utiliser le test if :



Il est donc logique de se dire que si ce chien n’a pas de nom, il ne doit pas être affiché.

Nous allons donc modifier le template principal en introduisant la logique if.

```
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    {{#if Nom}}
      <h2>Nom du chien : {{Nom}}.</h2>
      <p>Couleur du chien : {{Couleur}}.</p>
      {{> raceChien}}
    {{/if}}
  {{/each}}
</script>
```

La logique if utilise cette syntaxe pour la balise d’ouverture : `{{#if Nom}}`. Le mot-clé if est préfixé par le caractère # et est suivi par le nom de l’entrée sur laquelle se fait le test. Pour la balise de fermeture, nous avons la syntaxe `{{/if}}` : le mot-clé if est préfixé par le caractère /, sans qu’il soit nécessaire d’indiquer le nom de l’entrée.

Voici l’affichage obtenu :

Nom du chien : Raja.

Couleur du chien : Brun.

Race du chien : Labrador.

Nom du chien : Yuky.

Couleur du chien : Noir.

Race du chien : Caniche.

2. Le code complet

Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <div id="conteneur"></div>
    <script id="template" type="text/x-handlebars-template">
      {{#each chiens}}
        {{#if Nom}}
          <h2>Nom du chien : {{Nom}}.</h2>
          <p>Couleur du chien : {{Couleur}}.</p>
          {{> raceChien}}
        {{/if}}
      {{/each}}
    </script>
    <script id="race" type="text/x-handlebars-template">
      <p><em>Race du chien</em> : <strong>{{Race}}</strong>.</p>
    </script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/
4.0.11/handlebars.min.js"></script>
    <script>
      // Définir le bloc externe
      var laRace = document.getElementById('race').innerHTML ;
      Handlebars.registerPartial('raceChien',laRace) ;
      // Définir les données internes en JSON
      var chiensJSON =
{"chiens":[{"Nom":"Raja","Couleur":"Brun","Race":"Labrador"},
{"Couleur":"Beige","Race":"Beagle"}, {"Nom":"Yuky",
"Couleur":"Noir","Race":"Caniche"}]} ;
      // Définir et compiler le template
      var leTemplate = document.getElementById('template').innerHTML ;
      var compileTemplate = Handlebars.compile(leTemplate) ;
      // Exécuter le rendu
      var rendu = compileTemplate(chiensJSON);
      // Définir le conteneur
      var leConteneur = document.getElementById('conteneur') ;
      // Afficher le rendu
      leConteneur.innerHTML = rendu ;
    </script>
```

```
</body>
</html>
```

L'exemple à télécharger est dans le dossier **Chapitre-08-H/exemple-01.html**.

Utiliser un helper de contexte

1. Déterminer le contexte

Dans cet exemple, nous allons aborder le helper with, qui permet d'afficher des données d'un contexte spécifié. Si nous reprenons nos données sources avec l'ajout des propriétaires, Proprietaire, nous pouvons voir que ceux-ci sont insérés en tant qu'objets dans le tableau de données chiens.

```
var chiensJSON = {"chiens":[
  {"Nom":"Raja","Couleur":"Brun","Race":"Labrador",
    "Proprietaire": {"NomPro":"Durand","PrenomPro":"Paul"}
  },
  {"Nom":"Lord","Couleur":"Beige","Race":"Beagle",
    "Proprietaire": {"NomPro":"Leconte","PrenomPro":"Valérie"}
  },
  {"Nom":"Yuky","Couleur":"Noir","Race":"Caniche",
    "Proprietaire": {"NomPro":"Tralent","PrenomPro":"Julie"}
  }
]}
```

Si nous souhaitons afficher les données NomPro et PrenomPro avec cette syntaxe :

```
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    <p>Nom du chien : {{{gras Nom}}}.</p>
    <p>Couleur du chien : {{{gras Couleur}}}.</p>
    {{> raceChien}}
    <p>Le propriétaire : {{PrenomPro}} {{NomPro}}.</p>
  {{/each}}
</script>
```

Voici l'affichage obtenu :

Nom du chien : **Raja**.
Couleur du chien : **Brun**.
Race du chien : **Labrador**.
Le propriétaire : .
Nom du chien : **Lord**.
Couleur du chien : **Beige**.
Race du chien : **Beagle**.
Le propriétaire : .
Nom du chien : **Yuky**.
Couleur du chien : **Noir**.
Race du chien : **Caniche**.
Le propriétaire : .

Les données NomPro et PrenomPro ne sont pas affichées, car elles ne font pas partie du contexte de données chiens. Les données NomPro et PrenomPro font partie du contexte de données Proprietaire.

2. Le helper with

Avec le helper with, nous pouvons indiquer le contexte d'utilisation pour afficher des données imbriquées dans un autre tableau, dans un autre objet de données.

La syntaxe du helper with reprend celles que nous avons vues précédemment. La balise d'ouverture utilise le préfixe # devant le mot-clé with, suivi par le nom du contexte de données : {{#with Proprietaire}}. La balise de fermeture utilise le préfixe / devant le mot-clé : {{/with}}.

Voici le code du template principal :

```
<script id="template" type="text/x-handlebars-template">
  {{#each chiens}}
    <p>Nom du chien : {{{gras Nom}}}.</p>
    <p>Couleur du chien : {{{gras Couleur}}}.</p>
    {{> raceChien}}
    {{#with Proprietaire}}
      <p>Le propriétaire : {{PrenomPro}} {{NomPro}}.</p>
    {{/with}}
  {{/each}}
</script>
```

Voici l'affichage obtenu :

Nom du chien : **Raja**.
Couleur du chien : **Brun**.
Race du chien : **Labrador**.
Le propriétaire : Paul Durand.
Nom du chien : **Lord**.
Couleur du chien : **Beige**.
Race du chien : **Beagle**.
Le propriétaire : Valérie Leconte.
Nom du chien : **Yuky**.
Couleur du chien : **Noir**.
Race du chien : **Caniche**.
Le propriétaire : Julie Tralent.

3. Le code complet

Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <div id="conteneur"></div>
    <script id="template" type="text/x-handlebars-template">
      {{#each chiens}}
        <p>Nom du chien : {{{gras Nom}}}.</p>
        <p>Couleur du chien : {{{gras Couleur}}}.</p>
        {{> raceChien}}
        {{#with Proprietaire}}
          <p>Le propriétaire : {{PrenomPro}} {{NomPro}}.</p>
        {{/with}}
      {{/each}}
    </script>
    <script id="race" type="text/x-handlebars-template">
      <p><em>Race du chien</em> : {{{gras Race}}}.</p>
    </script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/
4.0.11/handlebars.min.js"></script>
    <script>
      // Définir le helper pour mettre en gras
      Handlebars.registerHelper('gras', function(texte){
        return '<strong>'+texte+'</strong>' ;
      })
      // Définir le bloc externe
      var laRace = document.getElementById('race').innerHTML ;
      Handlebars.registerPartial('raceChien', laRace) ;
    </script>
  </body>
</html>
```

```

// Définir les données internes en JSON
var chiensJSON = {"chiens":[
    {"Nom":"Raja","Couleur":"Brun","Race":"Labrador",
      "Proprietaire": {"NomPro":"Durand","PrenomPro":"Paul"}
    },
    {"Nom":"Lord","Couleur":"Beige","Race":"Beagle",
      "Proprietaire": {"NomPro":"Leconte","PrenomPro":"Valérie"}
    },
    {"Nom":"Yuky","Couleur":"Noir","Race":"Caniche",
      "Proprietaire": {"NomPro":"Tralent","PrenomPro":"Julie"}
    }
  ]} ;
// Définir et compiler le template
var leTemplate = document.getElementById('template').innerHTML ;
var compileTemplate = Handlebars.compile(leTemplate) ;
// Exécuter le rendu
var rendu = compileTemplate(chiensJSON);
// Définir le conteneur
var leConteneur = document.getElementById('conteneur') ;
// Afficher le rendu
leConteneur.innerHTML = rendu ;
</script>
</body>
</html>

```

L'exemple à télécharger est dans le dossier **Chapitre-08-I/exemple-01.html**.

Le stockage des données dans le navigateur

Dans le développement de vos sites web, vous pouvez avoir besoin de données qui soient accessibles et utilisables tout le temps. Nous pouvons bien sûr utiliser des champs de saisie pour que les utilisateurs puissent y saisir des informations, et par la suite nous pouvons accéder à ces informations. Mais le problème est que ces données ne sont plus exploitables une fois que la fenêtre ou l'onglet du navigateur sont fermés. Nous pouvons aussi recourir à des cookies, mais leur utilisation n'est pas très souple et ils sont limités en taille de stockage à 4 ko.

Pour avoir des données persistantes, le W3C nous fournit une API dédiée : le **Web Storage** : <https://www.w3.org/TR/webstorage/>

Web Storage nous met à disposition deux systèmes, deux interfaces (pour reprendre la terminologie officielle) pour stocker les données :

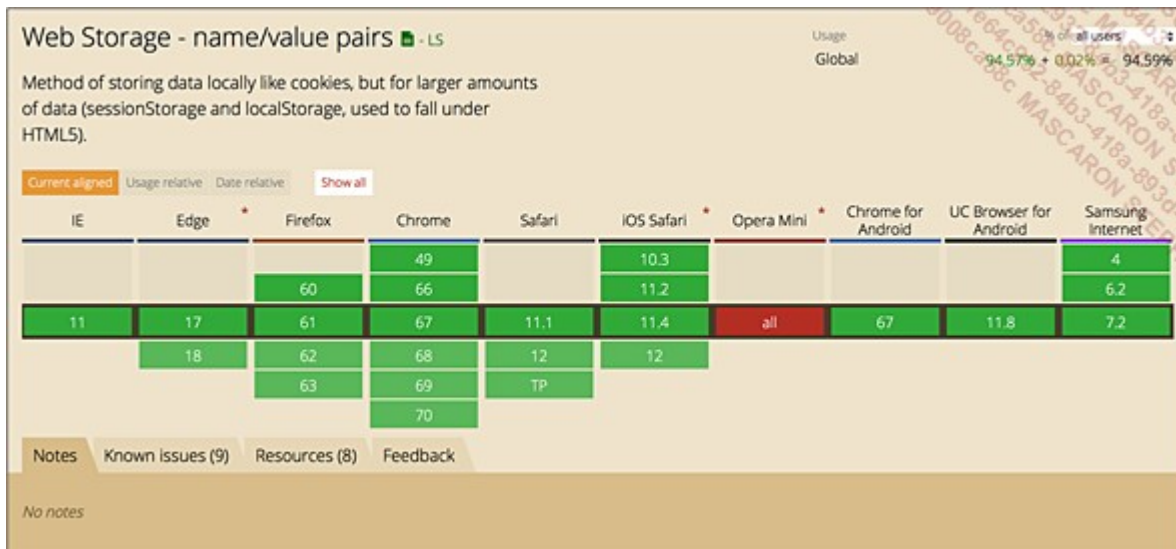
- `sessionStorage` permet de ne stocker les données que durant la session de la fenêtre ou de l'onglet du navigateur. Dès que vous fermez la fenêtre ou l'onglet, les données sont supprimées, donc elles ne sont plus accessibles.
- `localStorage` permet de stocker les données, même si vous fermez la fenêtre ou l'onglet et même si vous quittez le navigateur. Les données sont stockées localement jusqu'à 5 Mo, sans limite de durée de vie. De plus, les données sont accessibles à travers les différents onglets ouverts d'un même navigateur. Cela ne fonctionne pas avec plusieurs navigateurs ou avec des noms de domaine différents.

Cette technique offre de nombreux avantages : stockage rapide des données, les serveurs ne sont pas sollicités, pas de requêtes HTTP supplémentaires, pas besoin d'utiliser des cookies, les données sont accessibles même s'il n'y a pas de connexion Internet.



Attention, sachez que les données ne sont pas cryptées. Cela peut être un frein dans certains développements.

Dernier point important, le Web Storage est parfaitement reconnu par tous les navigateurs, comme vous le montre le site caniuse.com :



Les propriétés de Web Storage

1. Les interfaces

Nous venons de le voir, nous pouvons stocker des informations de manière temporaire avec sessionStorage et de manière permanente avec localStorage. Toutes les propriétés fonctionnent de la même manière avec ces deux systèmes (ces deux interfaces, pour reprendre la terminologie officielle).

L'exemple à télécharger est dans le dossier **Chapitre-09-B/exemple-01.html**.

2. Ajouter une donnée

Pour ajouter une donnée en stockage local, nous devons utiliser `setItem(key, value)`. Le premier argument est le nom de cette donnée. Le deuxième argument est la valeur de cette donnée.

Voici un exemple simple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <script>
      localStorage.setItem("Nom", "Aubry");
    </script>
  </body>
</html>
```



```
</body>
</html>
```

Ce script ne contient qu'une seule ligne : la création d'un enregistrement local d'une donnée nommée Nom et ayant pour valeur Aubry.

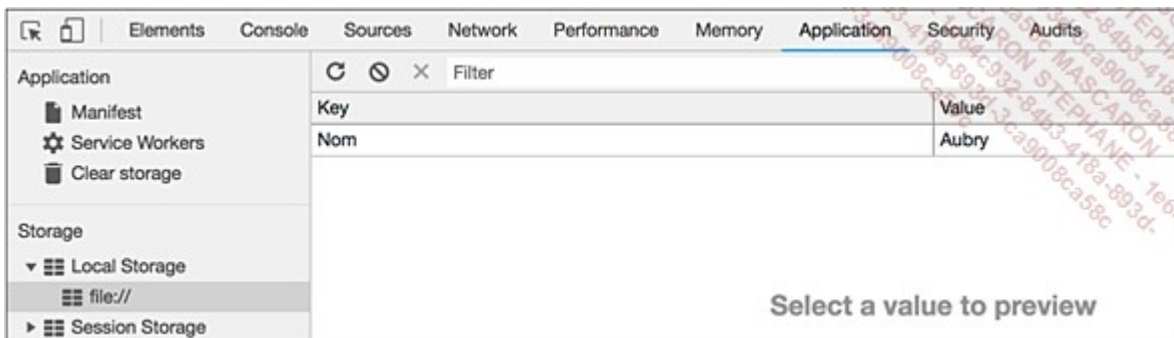
Affichez cette page dans un navigateur, Google Chrome dans cet exemple.

Affichez les outils pour les développeurs.

Cliquez sur l'onglet **Application**.

Dans le volet de gauche, dans la zone **Storage**, affichez **Local Storage** et cliquez sur **file://**.

Vous visualisez bien la donnée stockée :



Si vous fermez l'onglet, la fenêtre, ou si vous quittez le navigateur, à la prochaine ouverture les données stockées localement seront toujours accessibles.

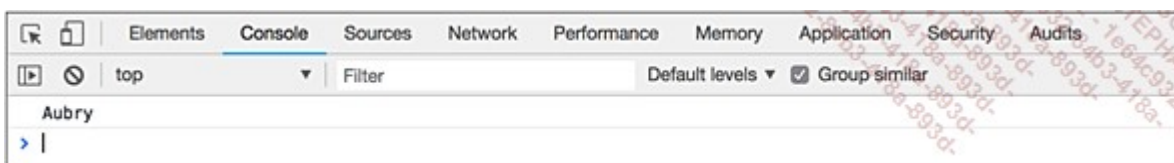
Vous pouvez aussi utiliser cette syntaxe pour enregistrer une donnée :

```
sessionStorage.Nom = "Aubry";
```

3. Récupérer une donnée

Pour récupérer une donnée stockée, il faut utiliser `session.getItem(key)`. Le paramètre `key` est le nom de la donnée que vous souhaitez récupérer.

Voici l'affichage obtenu dans la console :



Vous pouvez aussi utiliser cette syntaxe pour récupérer et afficher une donnée dans la console :

```
console.log(sessionStorage.Nom);
```

4. Supprimer des données

Pour supprimer une donnée stockée, il faut utiliser `session.removeItem(key)`. Le paramètre `key` est le nom de la donnée que vous souhaitez supprimer : `session.removeItem("Nom");`.

Voici un exemple avec un stockage local, `localStorage` :

```
<!DOCTYPE html>
```

```

<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <script>
      localStorage.setItem("Nom", "Aubry");
      var le_nom = localStorage.getItem("Nom");
      console.log(le_nom);
      localStorage.removeItem("Nom");
      console.log(sessionStorage.Nom);
    </script>
  </body>
</html>

```

Voici l’affichage obtenu dans la console :



La première ligne de la console affiche le résultat du premier `console.log(le_nom)` et affiche la valeur de la donnée Nom avec sa variable associée.

La deuxième ligne de la console affiche le résultat du deuxième `console.log(sessionStorage.Nom)`, et comme la donnée a été supprimée juste avant, nous avons bien logiquement l’affichage `undefined`.

Dans le cas où vous avez plusieurs données stockées à supprimer, utilisez : `sessionStorage.clear()`. Toutes les données seront alors supprimées.

La source des données

1. Saisir des données

Dans les exemples précédents, les données stockées localement étaient directement saisies dans les scripts. L’objectif était d’appréhender les fonctions natives de Web Storage. Bien sûr, dans la réalité des développements, cela n’est pas envisageable.

Dans l’exemple qui va suivre, nous allons utiliser un groupe de boutons radio et deux champs pour obtenir des données. L’utilisateur va saisir des informations que nous allons ensuite stocker localement.

Voici la structure de la page :

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <h3>Données Saisies</h3>
    <fieldset>

```

```

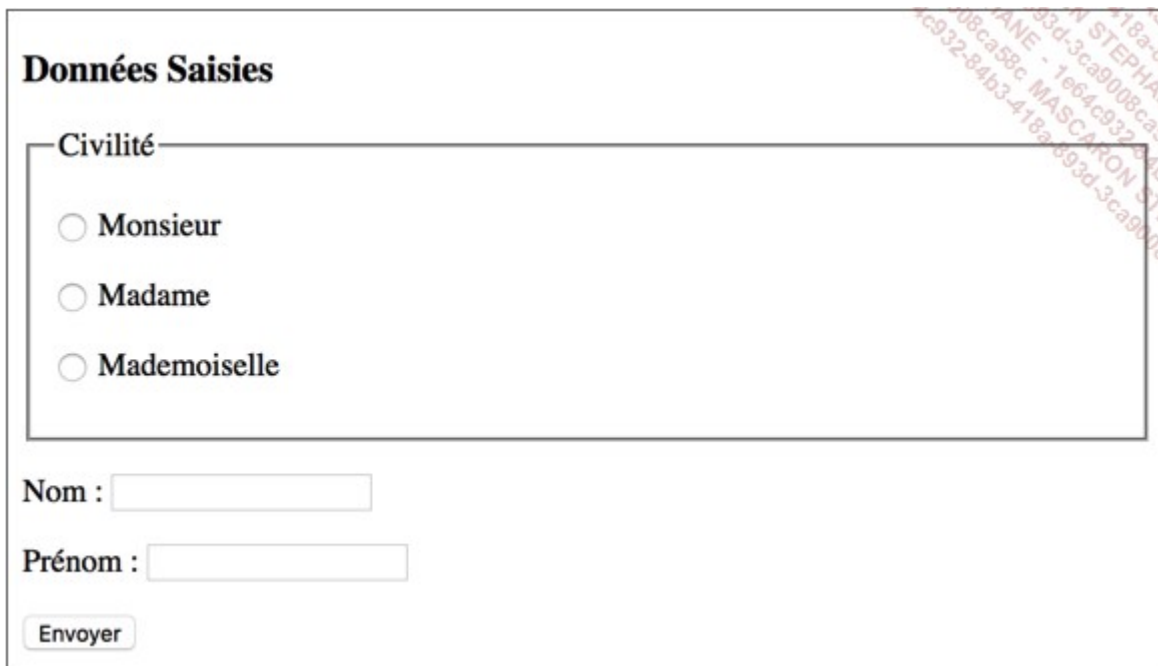
<legend>Civilité</legend>
<p>
name="civilite" <input type="radio" id="monsieur" value="Monsieur" />
<label for="monsieur">Monsieur</label>
</p>

<p>
name="civilite" <input type="radio" id="madame" value="Madame" />
<label for="madame">Madame</label>
</p>

<p>
value="Mademoiselle" name="civilite" <input type="radio" id="mademoiselle" />
<label for="mademoiselle">Mademoiselle</label>
</p>
</fieldset>
<p>
<label for="nom">Nom : </label>
<input id="nom" name="nom" type="text">
</p>
<p>
<label for="prenom">Prénom : </label>
<input id="prenom" name="prenom" type="text">
</p>
<p>
<button onclick="gererData()">Envoyer</button>
</p>
</body>
</html>

```

Voici l’affichage initial obtenu :



Données Saisies

Civilité

☐ Monsieur

☐ Madame

☐ Mademoiselle

Nom :

Prénom :

La structure de la page est très simple. Nous avons à notre disposition un groupe de boutons radio dont le nom commun est spécifié : `name="civilite"`. Ensuite, nous avons deux champs de saisie identifiés : `id="nom"` et `id="prenom"`. Nous allons bien sûr utiliser ces attributs dans le script. Enfin, lorsque l'utilisateur va cliquer sur le bouton, cela va exécuter une fonction nommée `gererData()`.

L'exemple à télécharger est dans le dossier **Chapitre-09-C/exemple-01.html**.

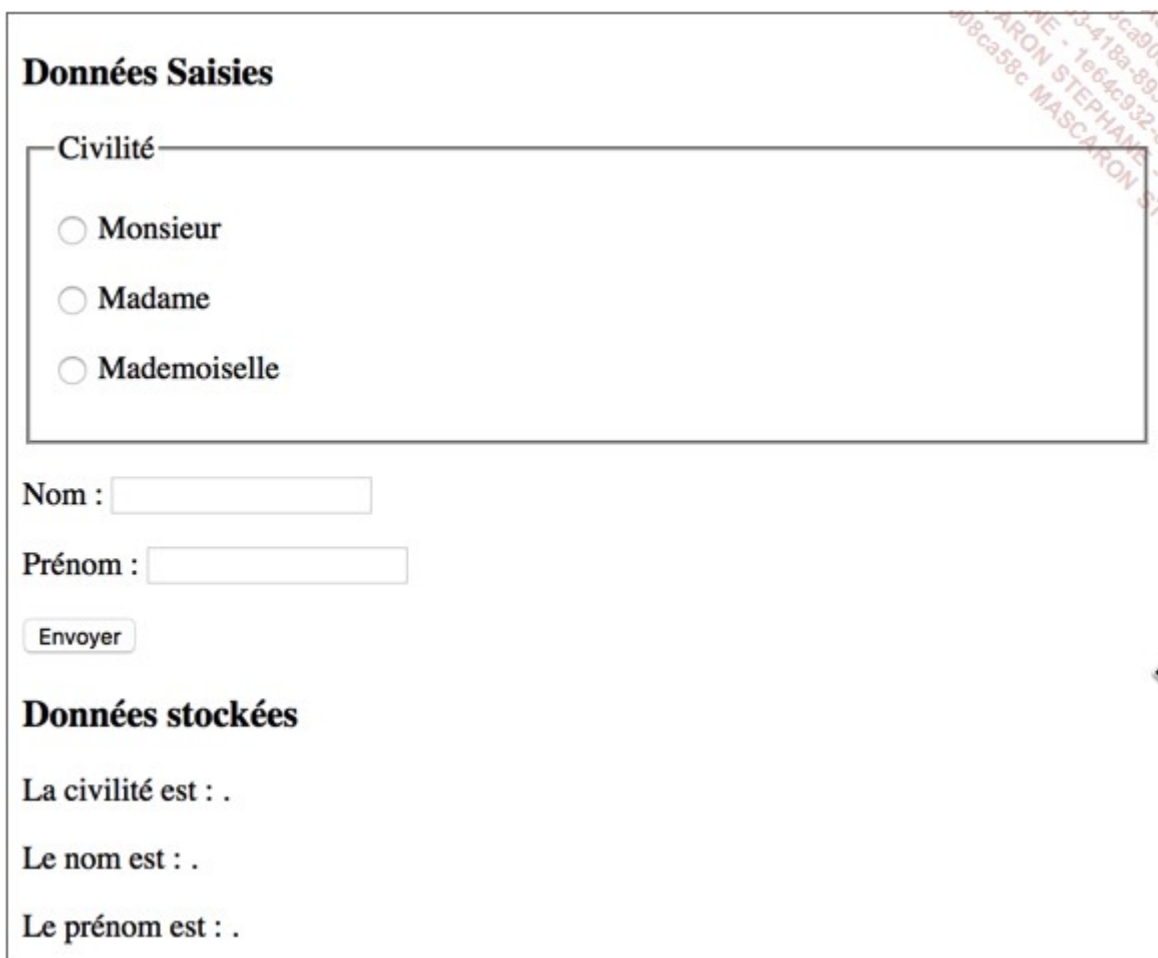
2. Gérer les données

L'objectif du script va être de récupérer les données saisies, de les stocker localement et de les afficher dans la page.

Pour afficher les données, à la suite du bouton d'envoi, nous créons trois paragraphes possédant trois éléments `` identifiés :

```
<h3>Données stockées</h3>
<p>La civilité est : <span id="laCivilite"></span>.</p>
<p>Le nom est : <span id="leNom"></span>.</p>
<p>Le prénom est : <span id="lePrenom"></span>.</p>
```

Voici l'affichage obtenu :



The screenshot shows a web form with two sections. The top section, titled "Données Saisies", contains a "Civilité" label followed by three radio buttons: "Monsieur", "Madame", and "Mademoiselle". Below these are input fields for "Nom" and "Prénom", and an "Envoyer" button. The bottom section, titled "Données stockées", displays the results of the form submission: "La civilité est : .", "Le nom est : .", and "Le prénom est : .". A diagonal watermark is visible across the top right of the form area.

Bien sûr, libre à vous de n'afficher ces éléments que lorsque le bouton d'envoi est cliqué.

Voyons maintenant le script :

```
<script>
/* Fonction de stockage sur le bouton */
function gererData() {
    /* Récupérer le bouton radio coché */
    var la_civilite ;
    var les_civilites=document.getElementsByName("civilite");
    for(i=0;i<les_civilites.length;i++) {
        if (les_civilites[i].checked==true) {
            la_civilite = les_civilites[i].value;
        }
    }
}
```

```

    }
  }
  /* Stocker les saisies */
  localStorage.setItem("Nom",
document.getElementById("nom").value);
  localStorage.setItem("Prénom",
document.getElementById("prenom").value);
  localStorage.setItem("Civilité", la_civilite);
  /* Récupérer et afficher les données */
  document.getElementById("leNom").innerHTML =
localStorage.getItem("Nom");
  document.getElementById("lePrenom").innerHTML =
localStorage.getItem("Prénom");
  document.getElementById("laCiviline").innerHTML =
localStorage.getItem("Civilité");
}
</script>

```

Détaillons ce script :

- Nous créons la fonction `gererData()` qui sera appelée lorsque l'utilisateur cliquera sur le bouton.

La première partie permet de récupérer la valeur du bouton radio sélectionné.

- Nous créons une variable `la_civilite` qui stockera la valeur du bouton radio sélectionné.
- Avec la fonction `getElementsByName("civilite")`, nous récupérons dans la variable `les_civilites` tous les éléments HTML dont le nom est `civilite`. Cela correspond bien aux boutons radio.
- Nous créons une boucle `for()` pour récupérer la valeur du bouton radio sélectionné dans la variable `la_civilite`.

La deuxième partie permet de stocker localement les données saisies.

- Avec `localStorage.setItem()`, nous stockons localement les données.
- Le premier argument des trois fonctions précédentes définit le nom de la valeur, respectivement `Nom`, `Prenom` et `Civilité`.
- Le deuxième argument permet de récupérer la valeur, `value`, des deux champs de saisie identifiés, respectivement `document.getElementById("nom").value` et `document.getElementById("prenom").value`. Et nous récupérons la valeur de la variable `la_civilite`.

La troisième partie permet de récupérer les données stockées localement et de les afficher.

- Nous utilisons `innerHTML`, appliqué à chaque élément HTML identifié pour afficher des données `document.getElementById("leNom")` `document.getElementById("lePrenom")` et `document.getElementById("laCiviline")`.
- Nous affichons les valeurs stockées localement avec `localStorage.getItem()`, pour chaque donnée : `localStorage.getItem("Nom")`, `localStorage.getItem("Prénom")` et `localStorage.getItem("Civilité")`.

Voici l'affichage obtenu :

Données Saisies

Civilité

☒ Monsieur

☐ Madame

☐ Mademoiselle

Nom :

Prénom :

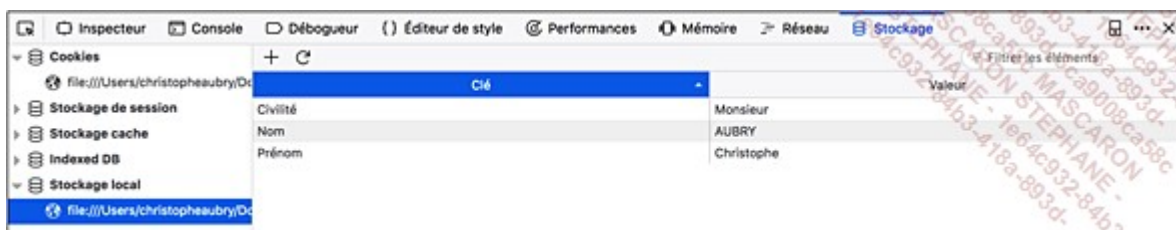
Données stockées

La civilité est : Monsieur.

Le nom est : AUBRY.

Le prénom est : Christophe.

Voici l’affichage obtenu dans les outils de développement de Mozilla Firefox :



3. Le code complet

Voici le code complet de cet exemple :

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Ma page</title>
  </head>
  <body>
    <h3>Données Saisies</h3>
    <fieldset>
      <legend>Civilité</legend>
      <p>
        <input type="radio" id="monsieur" value="Monsieur"
name="civillite" />
        <label for="monsieur">Monsieur</label>
      </p>
    </fieldset>
  </body>
</html>
```

```

        <p>
            <input type="radio" id="madame" value="Madame"
name="civillite" />
            <label for="madame">Madame</label>
        </p>

        <p>
            <input type="radio" id="mademoiselle"
value="Mademoiselle" name="civillite" />
            <label for="mademoiselle">Mademoiselle</label>
        </p>
    </fieldset>
    <p>
        <label for="nom">Nom : </label>
        <input id="nom" name="nom" type="text">
    </p>
    <p>
        <label for="prenom">Prénom : </label>
        <input id="prenom" name="prenom" type="text">
    </p>
    <p>
        <button onclick="gererData()">Envoyer</button>
    </p>
    <h3>Données stockées</h3>
    <p>La civilité est : <span id="laCivillite"></span>.</p>
    <p>Le nom est : <span id="leNom"></span>.</p>
    <p>Le prénom est : <span id="lePrenom"></span>.</p>
    <script>
        /* Fonction de stockage sur le bouton */
        function gererData() {
            /* Récupérer le bouton radio coché */
            var la_civillite ;
            var
les_civilites=document.getElementsByName("civillite");
            for(i=0;i<les_civilites.length;i++) {
                if (les_civilites[i].checked==true) {
                    la_civillite = les_civilites[i].value;
                }
            }
            /* Stocker les saisies */
            localStorage.setItem("Nom",
document.getElementById("nom").value);
            localStorage.setItem("Prénom",
document.getElementById("prenom").value);
            localStorage.setItem("Civilité", la_civillite);
            /* Récupérer et afficher les données */
            document.getElementById("leNom").innerHTML =
localStorage.getItem("Nom");
            document.getElementById("lePrenom").innerHTML =
localStorage.getItem("Prénom");
            document.getElementById("laCivillite").innerHTML =
localStorage.getItem("Civilité");
        }
    </script>
</body>
</html>

```